HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

DIPLOMARBEIT

Herr
Christian Schuster

Echtzeitfähiges Betriebssystem für STM32

Mittweida, 2017

Fakultät Ingenieurswissenschaften

DIPLOMARBEIT

Echtzeitfähiges Betriebssystem für STM32

Autor:
Herr

Christian Schuster

Studiengang:
Elektro- und Informationstechnik

Seminargruppe:
EI12wA-D

Erstprüfer:
Herr Prof. Dr.-Ing. Thomas Beierlein

Zweitprüfer:
Herr Dipl.-Ing Ronald Sieber

Einreichung:
Mittweida, 31.08.2017

Verteidigung/Bewertung:
Mittweida, 2017

Bibliografische Beschreibung:

Schuster, Christian:
Echtzeitfähiges Betriebssystem für STM32
2017 – 53 Seiten
Mittweida, Hochschule Mittweida (FH), University of Applied Sciences
Fakultät Ingenieurswissenschaften, Diplomarbeit, 2017

Referat:

Die vorliegende Arbeit befasst sich mit der Implementierung von FreeRTOS und lwIP auf einem Mikrocontroller vom Typ STM32F7. Für eine verschlüsselte Netzwerkkommunikation sollte dafür der vorhandene Crypto/Hash-Prozessor verwendet werden. Die erforderlichen theoretischen Grundlagen werden erläutert, ehe auf die Durchführung und den Ablauf eingegangen wird.

Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ASCII	american standard code of information interchange
API	application programming interface
ARP	address resolution protocol
CA	certification authority
CMSIS	cortex microcontroller software interface standard
DHCP	dynamic host configuration protocol
DMA	direct memory access
FreeRTOS	free real time operating system
HAL	hardware abstraction layer
HTTP(S)	hypertext transfer protocol (secure)
ICMP	internet control message protocol
IGMP	internet group management protocol
IoT	Internet of Things
IP, IPv4/6	internet protocol version 4/6
ISO	International Organization for Standardization
LwIP	lightweight TCP/IP stack
M2M	machine to machine
MLD	multicast listener discovery
MQTT	message queue telemetry transport
NVIC	nested vectored interrupt controller
OSI	open system interconnection
PHY	physical layer
PPP	point to point protocol
PPPoE	PPP over ethernet
PPPoS	PPP over serial

QR	quick response
RBU	receive buffer unavailable
RFID	radio-frequency identification
Rx	receiver
SMTP(S)	simple mail transfer protocol (secure)
SNTP	simple network time protocol
SoM	system on module
SSL	secure socket layer
TCP	transmisison control protocol
TLS	transport layer security
Tx	transmitter
UDP	user datagram protocol

1 Einleitung

Das Internet der Dinge (engl. Internet of Things, kurz. IoT) bezeichnet die Idee, dass alle Geräte über ein Netzwerk, meist über das Internet, verbunden sind und miteinander kommunizieren können. Es wird, vor allem in den letzten Jahren, zunehmend wichtiger für das alltägliche Leben, besonders auch für die Automatisierung und industrielle Anwendungen. Das Ziel des IoT ist es, dem Menschen unmerklich immer mehr bei seinen Tätigkeiten zu unterstützen, Arbeiten zu vereinfachen und generell das Leben und den Alltag zu erleichtern. [18]

1.1 Verwendung im Alltag

Im Privatgebrauch werden vor allem sogenannte Wearables genutzt - direkt in oder an Kleidungsstücken eingearbeitete eingebettete Computer oder Mikrocontroller. Die Anwendungen reichen dabei vom Schrittzähler, Pulsmessung bis zum Überwachen und der Auswertung der Schlafqualität. Dabei kommuniziert in der Regel ein Sensor, zum Beispiel Puls-, Bewegungs- und Temperatursensoren, oft kombiniert mit einem üblichen Kleidungsstück, wie etwa einer Armbanduhr oder den Schuhen, mit einem externen Gerät, in den meisten Fällen das Smartphone. Auf diesem Gerät, werden dann alle Daten gespeichert und bei Bedarf auch ausgewertet. In der nachfolgenden Abbildung ist eine Armbanduhr inklusive eines Pulsmessgerätes, als Beispiel für Wearables, zu sehen. [17]

Abbildung 1 - Armband – Pulsmessgerät [1]

1.2 Industrielle Anwendung

In der Industrie führt das IoT zu einer Vernetzung der Produktions- und Transportsysteme mit den Produkten selbst. So können mittlerweile Maschinen über Sensoren Daten auf-

nehmen und mit anderen Stationen teilen, um auf die Probleme oder Ereignisse während der Produktion zu reagieren. Grundlage dafür ist die sogenannte M2M (engl. Machine to Machine, Maschine zu Maschine) Kommunikation. Die Objekterkennung beruht oft auf der Nutzung von RFID-Chips und QR- bzw. Barcodes und die Vernetzung erfolgt drahtlos über GSM und GPRS. Dadurch lässt sich zu jedem Zeitpunkt der Produktion oder Lieferung der Zustand und Standort eines Produktes feststellen.

Ein Beispiel aus dem Alltag ist die Sendungsverfolgung bei Logistikunternehmen wie DHL und FedEx. Dabei wird jedem Paket eine einzigartige Sendenummer oder Code vergeben. Bei jedem Prozessschritt wird die Nummer gescannt und durch Vernetzung per Internet dem Endverbraucher zur Verfügung gestellt. [2]

1.3 Aufgabenstellung

Ziel dieser Diplomarbeit ist es, das freie embedded Betriebssystem "FreeRTOS" auf die SYSTEC-eigene STM32-basierte Referenzhardware zu portieren und hinsichtlich Ressourcenbedarf und Echtzeiteigenschaften zu bewerten. Für die Ausrichtung auf IoT-Anwendungen ist außerdem der freie TCP/IP-Stack "lwIP" in das Betriebssystem einzubinden. Die Portierung erfolgt zunächst auf einen Mikrocontroller des Typs STM32F407 und im Anschluss daran auf den neu produzierten IoT-Chip mit einem STM32F777.

Zur Unterstützung verschlüsselter Netzwerkkommunikation soll die Nutzung der prozessorinternen Krypto-Engine in Verbindung mit dem "lwIP" Stack erarbeitet und evaluiert werden.

Das Thema der Arbeit umfasst damit folgende Punkte:

- Einarbeitung in FreeRTOS, lwIP Stack und Krypto-Engine des STM32
- Portierung von FreeRTOS und lwIP Stack auf ein SYSTEC-eigens STM32 SoM
- Evaluierung der Realisierungsmöglichkeiten zur Nutzung der Krypto-Engine für verschlüsselte TCP-Kommunikation
- Bewertung der Laufzeitreduzierung für die Übertragung von Datenblöcken bei Nutzung der Krypto-Engine im Vergleich zur rein softwarebasierten Verschlüsselung
- Unter Nutzung der Krypto-Engine Erzeugen einer verschlüsselten Verbindung mit einem MQTT-Broker

2 Theoretische Grundlagen

Diese Arbeit befasste sich eingehend mit der Verwendung eines Echtzeitbetriebssystems und der Implementierung einer verschlüsselten Netzwerkkommunikation auf eine Referenzhardware vom Typ STM32. Deren wichtigste theoretische Grundlagen werden daher in diesem Kapitel erläutert.

2.1 Echtzeitbetriebssysteme

Echtzeitbetriebssysteme sind eine Art der Betriebssysteme, die dafür gedacht sind, Anwendungen in Echtzeit auszuführen. Der Begriff Echtzeit ist nicht zwangsläufig mit schneller Bearbeitung gleichzusetzen, sondern steht für eine rechtzeitige und vollständige Bearbeitung. Im Gegensatz zu üblichen Betriebssystemen wie Linux oder Windows, werden Rechenprozesse sofort und vollständig ausgeführt, wenn sie die erforderliche Priorität besitzen. Sie können jedoch von höher priorisierten Prozessen unterbrochen werden.

Sie verwalten dabei die Hardware-Ressourcen eines Computers, Mikroprozessors oder Mikrocontrollers und bietet dabei eine Multitasking-Umgebung, in der mehrere Tasks, parallel ablaufen können. Jedem Task muss dabei eine Priorität zugeordnet werden, anhand derer die Zuteilung der Bearbeitungszeit erfolgt. Als Task bezeichnet man Programmcode, der eigenständig oder auch als Schnittstelle zu anderer Software dienen kann.

Zur Verwaltung der Ressourcen werden sogenannte Semaphoren eingesetzt. Der Semaphor steht dabei für eine Ressource des Systems, auf die ein Task zugreifen möchte. So lang kein entsprechender Semaphor freigegeben wird, muss der Task warten, ehe er Zugriff auf die Ressource bekommt. [3] [4]

Für die Umsetzung eines Echtzeitbetriebssystem müssen bestimmte Faktoren erfüllt werden:

- Realisierung mehrerer unabhängiger Handlungsabläufe, auch als Nebenläufigkeit bezeichnet
- Einhaltung aller Fristen der Teilaufgaben, das bedeutet rechtzeitige Reaktion auf Ereignisse
- Aufteilung der Rechenzeit muss sinnvoll sein
- Korrektes Verhalten, bei gleichzeitigem Zugriff mehrerer Tasks auf gemeinsame Ressourcen

Nebenläufigkeit, oder auch Parallelität genannt, bedeutet, dass mehrere Operationen oder Befehle voneinander unabhängig und gleichzeitig ausgeführt werden können. Auf einem

herkömmlichen Ein-Kern-Prozessor ist dies nicht umsetzbar, die Operationen können dann nur scheinbar nebenläufig, das heißt quasi-parallel, bearbeitet werden. Eine reale Nebenläufigkeit ist nur mit einem Mehrkernprozessor zu erreichen, bei dem jeder Prozessor je eine Aufgabe übernimmt.

Um die erforderliche Nebenläufigkeit auf einem Ein-Kern-Prozessor zu erreichen, müssen sich alle Tasks die Rechenzeit teilen. Dadurch kommt es dazu, dass Tasks sich gegenseitig verdrängen müssen. Diese Verdrängung kann mit präemptiven und nicht-präemptiven Multitasking geregelt werden. [5]

Beim präemptiven Multitasking wird jeder Task sofort von von einem anderen mit höherer Priorität verdrängt. Das nicht-präemptiven Multitasking lässt jeden Task, auch mit niedriger Priorität, seine Arbeit zu Ende führen, bevor ein Wechsel erfolgt. [4]

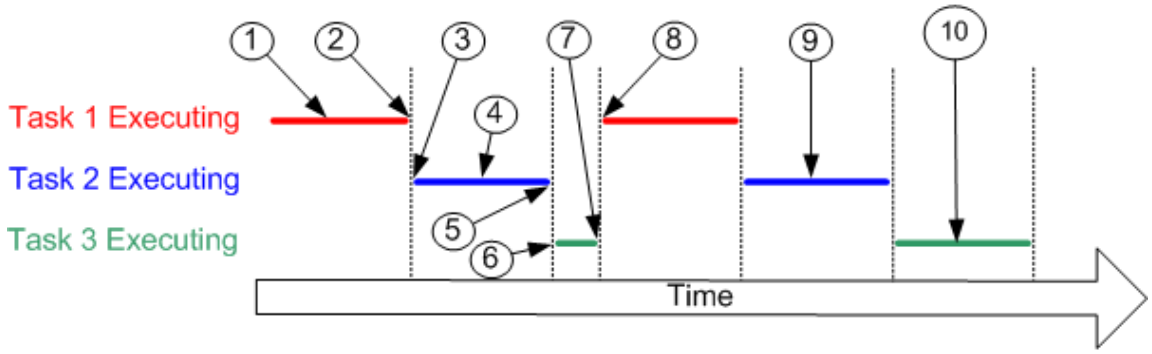

Abbildung 2 - Beispiel für Taskwechsel [6]

Der Wechsel zwischen Tasks wird vom Scheduler (dt. Zeitplaner) übernommen. Er ist „Bestandteil des Betriebssystems, das dem Prozessor die Aufgaben zuteilt. Er dient der Überwachung und Zuweisung der Systemressourcen sowie der Kontrolle aller ablaufenden Tasks.“ [32]

2.2 Kommunikationsprotokolle

Um sich einen Überblick über die vorhandenen Kommunikationsprotokolle zu verschaffen zieht man häufig das OSI-Modell heran. Es wurde von der ISO, der Internationalen Organisation für Standardisierung, entwickelt und soll als Grundlage für die Bildung von Kommunikationsstandards dienen. Das OSI-Modell ist in sieben Klassen, auch oft als Schichten bezeichnet, aufgeteilt. Je höher die Schicht, desto mehr wird vom eigentlichen Datentransport abstrahiert. In Abbildung 4 ist der Aufbau des Modells dargestellt.

Die übergeordneten Schichten bauen dabei in ihrer Funktion immer auf den unteren auf. So verarbeitet Schicht 2 Daten aus Schicht 1 und Schicht 3 aus 2. [7]

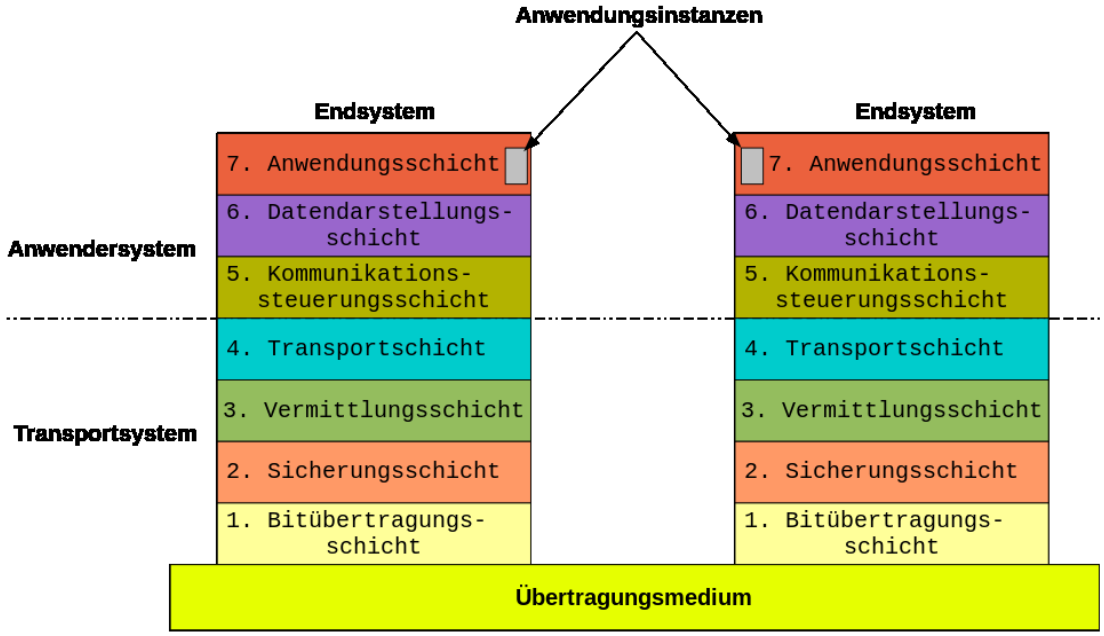

Abbildung 3 - OSI-Modell [8]

In der folgenden Tabelle werden die verschiedenen Protokolle, Übertragungs- und Vermittlungstechniken den Schichten des OSI-Modells zugeordnet.

Schicht 7	**Anwendung**	Telnet, FTP, HTTP, SMTP, NNTP
Schicht 6	**Darstellung**	Telnet, FTP, HTTP, SMTP, NNTP, NetBIOS
Schicht 5	**Kommunikation**	Telnet, FTP, HTTP, SMTP, NNTP, NetBIOS, TFTP
Schicht 4	**Transport**	TCP, UDP, SPX, NetBEUI
Schicht 3	**Vermittlung**	IP, IPX, ICMP, T.70, T.90, X.25, NetBEUI
Schicht 2	**Sicherung**	LLC/MAC, X.75, V.120, ARP, HDLC, PPP
Schicht 1	**Übertragung**	Ethernet, Token Ring, FDDI, V.110, X.25, Frame Relay, V.90, V.34, V.24

Tabelle 1 - OSI-Modell Protokolle [7]

Im Rahmen dieser Arbeit werden ausschließlich die verwendeten Schichten eins bis vier näher erklärt.

2.2.1 Übertragungsschicht

Auch oft als Bitübertragungsschicht bezeichnet, wandelt Schicht 1 Signale aus dem Übertragungskanal in Binärsignale um. Jeder Übertragungskanal (z.B. Ethernet) muss daher ein Gerät an beiden Enden besitzen, das Schicht-1-Protokolle unterstützt. Für die Kommunikation über das Internet gibt es kein spezielles Protokoll, so dass jedes verfügbare Protokoll benutzt werden kann. [9, S. 366 ff.]

2.2.2 Sicherungsschicht

Schicht 2 übernimmt die Daten aus Schicht 1 und überprüft deren Integrität und Richtigkeit mit Prüfsummen. Außerdem werden die Daten zu Bitgruppen zusammengefasst (z.B. ASCII-Zeichen). Schicht-2-Protokolle sind wie bei der Übertragungsschicht meist direkt am Geräteende vorhanden und hängen damit direkt von der Hardware ab. Dort ist auch bereits ein TCP/IP-Protokoll angesiedelt: das Point-to-Point-Protocol. Es wird dafür verwendet, eine feste Verbindung zwischen zwei Punkten zu erstellen. [9, S.366 ff.]

2.2.3 Vermittlungsschicht

In der Vermittlungsschicht werden die aus der Sicherungsschicht erhaltenen Zeichen weitergeleitet. Schicht-3-Protokolle kommen hauptsächlich in Netzknoten und Routern zum Einsatz. Die wichtigsten und am häufigsten verwendeten Protokolle sind IP, ARP, RARP, IMGP und IMCP. [9, S. 366 ff.]

2.2.4 Transportschicht

Auf Grundlage der Schicht 3 wird mit der Transportschicht eine Verbindung zwischen einem Sender und Empfänger von Daten erzeugt. Dabei werden zwischengeschaltete Router nicht bemerkt. Daher werden Schicht-4-Protokolle häufig von Netzknoten abgearbeitet, die Daten auch selbst verwenden. TCP und UDP sind zwei der bekanntesten und am meisten verwendeten Protokolle. Sie nutzen beide IP und stellen eine Ende-zu-Ende Verbindung her. [9, S. 366 ff.]

2.3 TCP/IP

TCP/IP ist der Sammelbegriff für eine Protokollfamilie für die Netzwerkkommunikation. Die bekanntesten und am meisten verwendeten Protokolle, sind die namensgebenden TCP und IP Protokolle, dazu gehören unter anderem auch UDP als weiteres Transportprotokoll und DHCP und ARP als Hilfs- bzw. Anwendungsprotokolle. [10]

2.3.1 IP

IP steht für Internet Protocol und stellt die Grundlage für die moderne Kommunikation über das Internet. Die Übertragung erfolgt auf der Vermittlungsschicht, ist paketorientiert und verbindungslos, das heißt, es wird keine Verbindung zwischen den Kommunikationspartnern etabliert und direkt mit dem Senden von Daten begonnen. Daher kann das IP-Protokoll nicht garantieren, dass alle Pakete in vorgesehener Reihenfolge ankommen oder nicht verloren gehen, zum Beispiel durch Netzwerküberlastungen.

Jeder individuelle Kommunikationsteilnehmer wird durch eine IP-Adresse erreichbar. Diese Adresse kann einem einzelnen Empfänger oder aber auch einer Gruppe von Empfänger zugeordnet sein. [11]

Die klassische IPv4-Adresse besteht aus 32 Bits, also vier Oktetten, voneinander mit einem Punkt getrennt, zum Beispiel 192.168.6.1. Damit sind 4.294.967.296 individuelle Adressen erstellbar.

Dadurch, dass der Bedarf an IP-Adressen in den letzten Jahren enorm angestiegen ist, wurde es notwendig, den vorhandenen Adressraum zu erweitern. Daher wurde IPv6 entwickelt. IPv6 verwendet 128 Bits, damit sind 2^{128} (= 340.282.366.920.938.463.463.374.607.431.768.211.456 ≈ $3{,}4 \cdot 10^{38}$) Adressen darstellbar. Eine Dezimaldarstellung dieser Adressen wäre sehr unübersichtlich und praktisch sehr schlecht handhabbar, daher wird sie hexadezimal dargestellt. Für weitere Vereinfachungen, werden je zwei Oktetten zusammengefasst und in Gruppen durch Doppelpunkt voneinander getrennt dargestellt, außerdem werden Nullen am Beginn einer Oktette weggelassen bzw. mehrere aufeinander folgende Blöcke, die nur aus Nullen bestehen mit doppeltem Doppelpunkt ersetzt. Zum Beispiel wird aus 2001:0db9:85a3:0000:0000:8a2e:0370:7344 durch Vereinfachung 2001:db9:85a3::8a2e:37:7344.

Es ist aber auch möglich, mehrere Teilnehmer mit einer Adresse zu erreichen. Dafür gibt es die Möglichkeit der Mehrpunktverbindung, zum Beispiel Broadcast und Multicast. Broadcast sendet Datenpakete an alle Teilnehmer eines Netzwerkes, welche die dafür notwendige Empfangsmöglichkeit haben, während bei Multicast vorher eine Anmeldung beim Sender erforderlich ist. [12]

2.3.2 TCP

TCP steht für Transmission Control Protocol (dt. Übertragungssteuerungsprotokoll). Es definiert, auf welche Art und Weise Daten zwischen Netzwerkkomponenten ausgetauscht werden. Anders als bei dem verbindungslosen IP und UDP stellt TCP eine Verbindung zwischen zwei Netzwerkpunkten her. Über diese Verbindung können in beide Richtungen Daten übertragen werden. Für diese Umsetzung setzt TCP auf das IP (Internet-Protokoll) auf, daher spricht man auch oft von dem “TCP/IP-Protokoll”. Die Verwendung von TCP bietet viele Vorteile für die praktische Anwendung, so erkennt es Datenverluste und ver-

sucht diese automatisch zu beheben, die Datenübertragung funktioniert in beide Richtungen, Netzwerküberlastungen werden verhindert usw. Deshalb ist es weit verbreitet und wird mittlerweile vom Großteil aller modernen Computer und Geräte für den Datenaustausch verwendet. [13]

2.3.2.1 *Verbindungsaufbau*

Typischerweise läuft der Verbindungsaufbau bei TCP über einen Drei-Wege-Handschlag (Three-Way-Handshake) ab. Ein Client schickt einen Verbindungswunsch (SYN) an einen Server seiner Wahl. Sofern der Server einen Verbindungsaufbau gestattet, reagiert dieser mit einer Bestätigung (ACK) und sendet ebenfalls einen Verbindungswunsch (SYN). Der Client bestätigt den Verbindungsaufbau ebenfalls (ACK). Im Anschluss daran kann der Datenaustausch erfolgen. Eine Darstellung dieses Ablaufes sieht man in der nachstehenden Abbildung. [13]

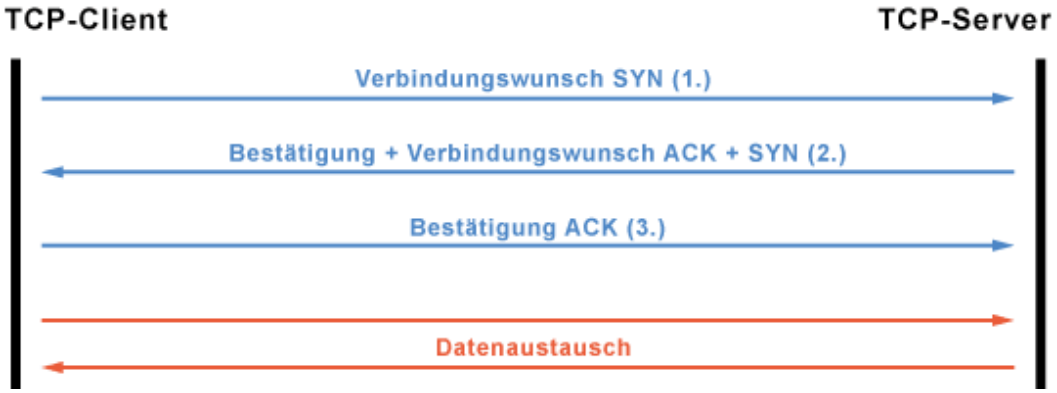

Abbildung 4 - Drei-Wege-Handschlag [14]

2.3.2.2 *Datenaustausch*

Der Ablauf beim Datenaustausch ist, wie der Verbindungsaufbau, in den meisten Fällen gleich. Der Sender schickt ein Paket zum Empfänger, welcher mit einer Bestätigung (ACK) den Empfang quittiert. Erst danach schickt der Sender das nächste Paket, welches wiederum vom Empfänger bestätigt wird. Dieser Prozess läuft solange weiter, bis alle Daten übertragen wurden. Für den Fall, dass für ein Paket keine Empfangsbestätigung eintrifft, ist bei jeder Übertragung ein Timer gesetzt, der bei Ablauf für das erneute Senden des Pakets sorgt. Dadurch wird ein möglicher Datenverlust ausgeschlossen. Dieser Prozess ist in der nachfolgen Abbildung grafisch veranschaulicht. [13]

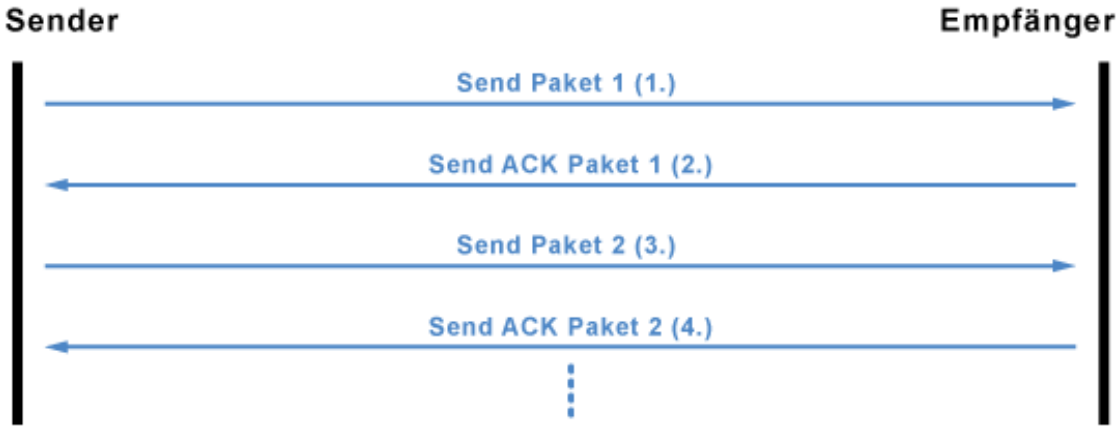

Abbildung 5 - Datenaustausch TCP [15]

2.3.2.3 *Verbindungsabbau*

Das Schließen bzw. Abbauen einer Verbindung läuft im Prinzip wie ein Verbindungsaufbau ab. Sowohl der Client, als auch der Server können einen Verbindungsabbauwunsch senden (FIN). Die Gegenstelle bestätigt den Erhalt (ACK) und antwortet ebenfalls mit einem Verbindungsabbauwunsch (FIN), der wiederum mit einer Bestätigung quittiert wird (ACK). Danach ist die Verbindung abgebaut. In der nachfolgenden Abbildung sieht man den Prozess zum Verbindungsabbau. [13]

Abbildung 6 - Verbindungsabbau TCP [16]

2.4 Verschlüsselung mit SSL/TLS

„Unter Verschlüsselung versteht man Verfahren und Algorithmen, die Daten mittels digitaler bzw. elektronischer Codes oder Schlüssel inhaltlich in eine nicht lesbare Form umwandeln. […] Gleichzeitig wird dafür gesorgt, dass nur mit dem Wissen eines Schlüssels die geheimen Daten wieder entschlüsselt werden können.
Anstatt von Verschlüsselung spricht man auch von Chiffrierung, was das gleiche meint.“ [31]

SSL (engl. für Secure Socket Layer) ist die ältere Bezeichnung für das heute verwendete TLS (engl. für Transport Layer Security) und funktioniert auf Basis von TCP. Im Wesentlichen bezeichnen beide Namen dasselbe Verfahren. Man entschied sich ab SSL Version 3.0 den Namen auf TLS beginnend mit der Version 1.0 zu ändern. Häufig wird der Begriff SSL benutzt, obwohl eigentlich TLS gemeint ist. Es ist ein hybrides Verschlüsselungsverfahren für die sichere Kommunikation im Internet. Bekannt ist es durch die Verschlüsselung des Web-Datenverkehrs mit dem Protokoll HTTPS.

SSL und TLS greifen zwar nicht direkt in die TCP-Kommunikation ein, sie werden trotzdem in der Transportschicht eingeordnet. Diese Zuordnung ist weiterhin eindeutig durch den Namen TLS (dt. Transportschicht-Sicherheit) begründet. [9, S. 401 ff.]

2.4.1 Zertifikate und Schlüssel

Ein Zertifikat dient dazu, sich im Internet gegenüber anderen als derjenige auszuweisen, für den man sich ausgibt. Ausgestellt werden Zertifikate von einer Zertifizierungsinstanz, auch als Certification Authority bezeichnet. Sehr wichtig ist, dass diese CA unabhängig und vertrauenswürdig ist, denn Zertifikate können prinzipiell auch selbst erzeugt werden. Programme wie zum Beispiel OpenSSL ermöglichen es, Zertifikate selber zu erstellen und zu signieren. Bei der Überprüfung wird daher auf Zertifikataussteller geschaut, um zu verifizieren, dass das Zertifikat aus einer vertrauensvollen Quelle stammt.

Beim Public-Key-Verfahren werden meist Zertifikate nach dem X.509 Standard verwendet. Sie beinhalten neben dem öffentlichen Schlüssel unter anderem auch die unterstützten TLS-Versionen, Signaturalgorithmen, Gültigkeitsdauer und auch den Aussteller des Zertifikats. Die Struktur eines Zertifikats, die Informationen die es beinhaltet und wie es für eine Website benutzt wird, ist in der folgenden Abbildung zu sehen.

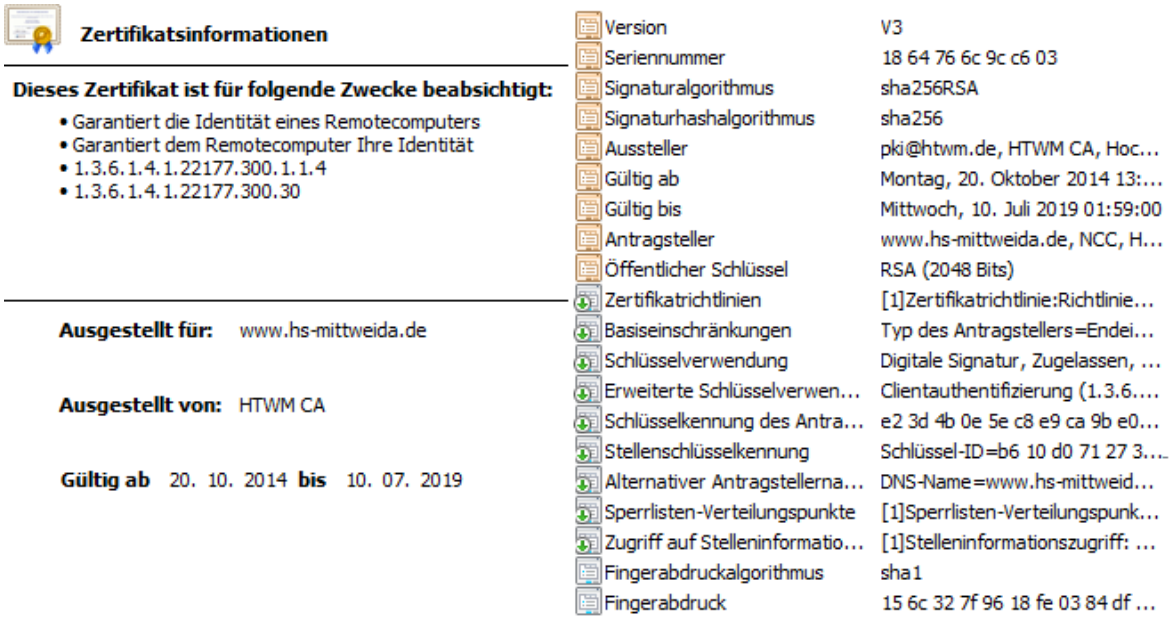

Abbildung 7 – X.509 Zertifikat der HS Mittweida

An die Zertifikatausstellung ist auch die Ausgabe eines privaten Schlüssels gebunden. Im Gegensatz zum öffentlichen Schlüssel, sollte sich dieser jedoch von keinem, außer dem Eigentümer einsehen lassen.

Beide Schlüssel sind durch mathematische Algorithmen miteinander verbunden. Dabei können mit dem öffentlichen Schlüssel kodierte Nachrichten ausschließlich mit dem privaten Schlüssel dekodiert werden. Dadurch ist es möglich, dass jeder Kommunikationsteilnehmer den öffentlichen Schlüssel erfahren darf, damit verschlüsselte Nachrichten an den Empfänger geschickt werden können und ausschließlich der Eigentümer des privaten Schlüssels in der Lage ist, diese zu entschlüsseln. [9, S. 279 ff.]

2.4.2 Verschlüsselungsverfahren

Verwendete Verschlüsselungsverfahren werden in drei verschiedene Kategorien aufgeteilt: symmetrisch, asymmetrisch und hybride Verfahren.

Symmetrische Verfahren ver- und entschlüsseln die Daten mit demselben Schlüssel. Dadurch wird das Verfahren recht sicher. Allerdings haben sie den großen Nachteil, dass der Schlüssel zunächst auf sichere Weise übertragen werden muss. Dieses Problem bezeichnet man oft auch als das Schlüsselaustauschproblem. In der folgenden Abbildung ist der Ablauf des symmetrischen Verfahren grafisch dargestellt. [19]

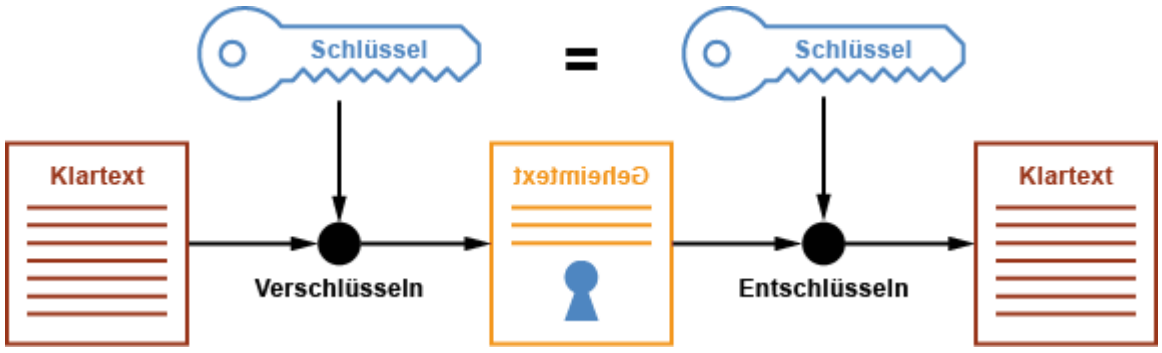

Abbildung 8 - Symmetrische Verschlüsselung [20]

Asymmetrische Verfahren nutzen nicht nur einen Schlüssel, sondern arbeiten mit einem Schlüsselpaar, dem öffentlichen und privaten Schlüssel. Deshalb nennt man diese Verfahren häufig auch Public-Key-Verfahren. Nachrichten oder Daten, die man nun mit dem öffentlichen Schlüssel verschlüsselt, können nur mit dem privaten Schlüssel entschlüsselt werden. Dadurch wird das Schlüsselaustauschproblem komplett gelöst. Die folgende Abbildung veranschaulicht den Verlauf. [21]

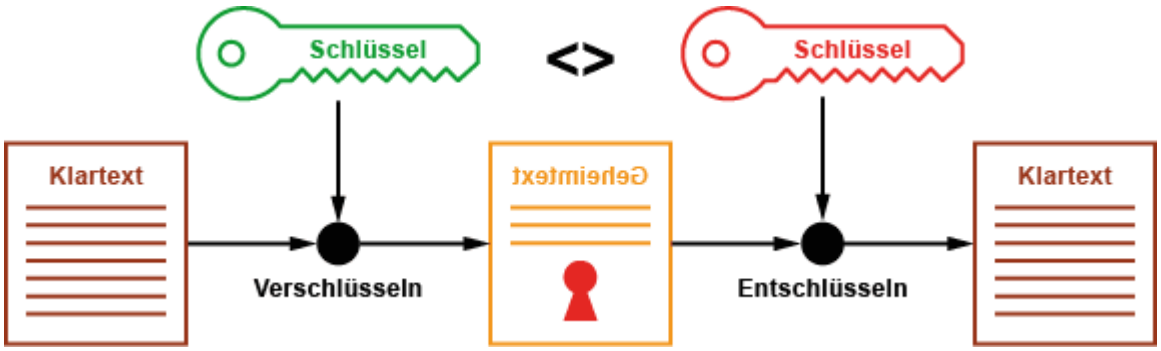

Abbildung 9 - Asymmetrische Verschlüsselung [22]

Hybride Verschlüsselungsverfahren kombinieren die Vorteile von symmetrischen und asymmetrischen Verfahren und lösen die individuellen Nachteile gleichzeitig auf. Zuerst wird ein zufälliger Sitzungsschlüssel generiert für die symmetrische Datenverschlüsselung. Dieser wird im Anschluss mit dem öffentlichen Schlüssel des Empfängers vom Sender verschlüsselt versendet und vom Empfänger mit seinem privaten Schlüssel entschlüsselt. Im Anschluss daran, können dann Daten zwischen beiden Parteien über den Sitzungsschlüssel symmetrisch verschlüsselt und ausgetauscht werden. Das Prinzip dafür wird in der untenstehenden Abbildung verdeutlicht. [23]

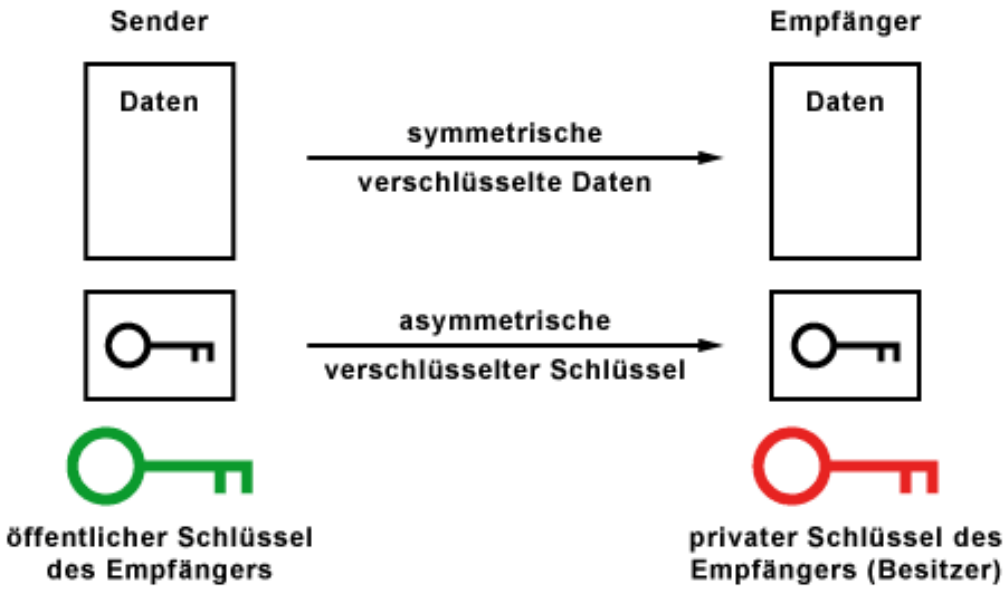

Abbildung 10 - Hybride Verschlüsselung [24]

2.4.3 TLS Verbindungsaufbau

Genau wie bei TCP beginnt der Aufbau einer Verbindung damit, dass ein Client sich per TCP mit einem Server verbinden möchte. Nach dem Verbindungsaufbau wird vom Client eine Liste der unterstützten Verschlüsselungsmethoden (Cipher Suites) an den Server mitgeschickt.

Daraufhin schickt der Server dem Client sein Zertifikat, inklusive öffentlichem Schlüssel, und eine von ihm ausgewählte und vom Client unterstütze Verschlüsselungsmethode. Die Auswahl einer geeigneten Verschlüsselungsmethode erfolgt auf Grundlage der vom Client geschickten Liste an den Server. Es wird die am stärksten verfügbare Verschlüsselung ausgewählt, die beide Teilnehmer unterstützen. Der Client überprüft das Serverzertifikat und authentisiert sich optional selbst auch noch gegenüber dem Server. Diesen Schritt nennt man auch TLS-Handshake. Ein schematischer Ablauf ist in der folgenden Abbildung zu sehen. [44]

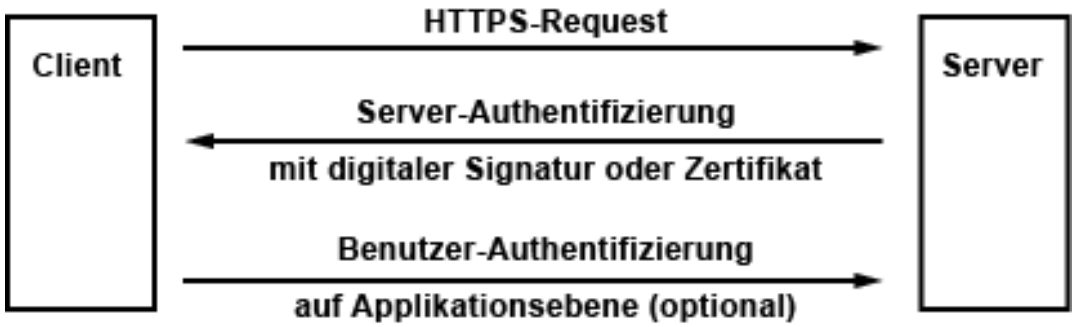

Abbildung 11 - Verbindungsablauf SSL/TLS [43]

Im Anschluss daran, kann mit dem verschlüsselten Datenaustausch begonnen werden.

3 Stand der Technik

In diesem Kapitel wird auf die verwendete Technik und Funktionsbibliotheken eingegangen. Bedingt durch die Anforderungen und Rahmenbedingungen der Aufgabenstellung und zukünftige Verwendung in weiteren Projekten sind der verwendete Mikrocontroller, das Echtzeitbetriebssystem und der IP-Stack vorgegeben.

3.1 STM32

STM32 ist der Oberbegriff für eine Mikrocontroller-Familie von STMicroelectronics, basierend auf den von ARM entwickelten Cortex-M Prozessoren. Diese Mikrocontroller wurden dazu entwickelt, dem Anwender neue Möglichkeiten und mehr Freiheit bei der Programmierung zu geben, als es bisher der Fall war.

Sie bieten eine sehr hohe Performance, Echtzeitfähigkeit, digitale Signalbearbeitung und energiesparende Operationen. Den STM32 gibt es in unzähligen Varianten, mit unterschiedlicher Peripherie, verschiedenen Größen und Ausstattungen.

Der verwendete Mikroprozessor für die SYSTEC-eigene Referenzhardware, zu sehen in der folgenden Abbildung, ist vom Typ STM32F777 und verwendet einen Cortex-M7 Kern, der mit bis zu 216 MHz läuft.

Abbildung 12 - IoT Chip [46]

Was den STM32F777 besonders von vergleichbaren MCUs abhebt, ist der integrierte Crypto/Hash-Prozessor für die Hardwarebeschleunigung von AES-128, -192 und -256 Verschlüsselungen. Ebenso unterstützt er Algorithmen wie GCM, CCM, TDES und Hash (MD5, SHA-1, SHA-2). [25] [26]

3.2 FreeRTOS

FreeRTOS wurde als Betriebssystem gewählt, da es das am weitesten verbreitete und am meisten verwendete Echtzeitbetriebssystem ist. Dadurch gibt es umfangreiche Dokumentationen, viele Anwendungsbeispiele und Software, unter anderem lwIP, die gezielt für den Einsatz mit FreeRTOS entwickelt wurden und so die Implementierung weiterer Funktionalitäten erleichtert.

Durch seine geringe Größe ist es besonders für eingebettete Systeme geeignet. Es ist Open-Source und kann in jeder Applikation, ob kommerziell oder privat ohne Einschränkung eingesetzt werden. Besonders hervorstechend sind der simple Aufbau, die hohe Verlässlichkeit, umfangreiche Features und der minimale Speicherverbrauch. [27]

3.3 LwIP - A lightweight TCP/IP stack

LwIP ist eine leichtgewichtige und unabhängige Implementation von TCP/IP Protokollen, ursprünglich entwickelt von Adam Dunkels. Damit lwIP auch in eingebetteten Systemen eingesetzt werden kann, liegt der Fokus der Weiterentwicklung immer darauf, die verwendeten (Hardware-)Ressourcen zu reduzieren bzw. gering zu halten, ohne dabei die Funktionalität einzuschränken.

Einige der wichtigsten Funktionen die lwIP bietet, sind unter anderem:

- Protokolle: IP, IPv6, ICMP, ND, MLD, UDP, TCP, IGMP, ARP, PPPoS, PPPoE
- DHCP Client, DNS Client, Auto-IP
- APIs: spezialisierte APIs für verbesserte Performance, optional Berkley-ähnliche Sockets
- Erweiterte Features: IP-Forwarding über mehrere Netzwerkinterfaces
- Zusätzliche Applikationsmöglichkeiten: HTTP(S) Server, SNTP Client, SMTP(S) Client, ping, MQTT Client [28]

3.3.1 BSD-Sockets

Besonders hervorzuheben ist die Möglichkeit zur Verwendung von Sockets. Auch als Sockel bezeichnet, bilden sie einen Kommunikationsendpunkt, um mit anderen Programmen Daten auszutauschen. Sockets haben den großen Vorteil, dass sie plattformunabhängige und standardisierte Schnittstellen sind. Es gibt vier Varianten, wie man einen Socket verwenden kann: als Client- oder Server-Socket und als Datagram- oder Stream-Socket. [45]

Nachfolgend sind die wichtigsten Funktionen der Socket-API aufgelistet:

socket() Diese Funktion erstellt einen Socket mit vorher definierten Eigenschaften (Datagram oder Stream). Um den Socket identifizieren zu können, wird eine eindeutige Zahl vom Typ Integer zurückgegeben.

bind() Damit wird ein Socket an eine feste IP-Adresse und Port gebunden. Dies wird in der Regel meist serverseitig benutzt.

listen() Ein Stream-Socket wird damit in einen Zuhören-Modus versetzt und wartet auf einkommende Verbindungsanfragen. Ebenso wie bind() wird dies meist nur von Servern benutzt.

connect() Connect versucht über IP-Adresse und Port, einen Socket mit einem anderen zu verbinden. Dies wird meist von einem Client für die Verbindung zu einem Server benutzt.

accept() Mit dieser Funktion wird eine eingehende TCP/IP-Verbindungsanfrage (siehe connect) bestätigt und ein neuer Socket für diese Verbindung erzeugt. Dieser neue Socket ist nur dem Server bekannt und dient der Kommunikation mit dem Client.

send()/recv() Diese Funktionen schreiben bzw. lesen Informationen von Sockets. Weiterhin gibt es write(), read(), sendto(), recvfrom() mit unterschiedlichem Funktionsumfang.

close() Close schließt entsprechende Sockets und TCP/IP-Verbindungen werden dadurch beendet.

Die Erstellung eines Sockets erfolgt immer nach den gleichen Prinzipen. Eine Client-Server-Verbindung wird im Wesentlichen in vier Schritten ausgeführt:

1. Art des Sockels festlegen und erstellen
2. Verbindungsaufbau mit einem Server (Adresse und Portnummer)
3. Senden und Empfangen von Daten
4. Falls nötig Verbindung trennen und Socket schließen

Im Gegensatz dazu, erfolgt die Erzeugung eines Sockels aus der Sicht des Servers in sechs Schritten, ähnelt aber stark dem obigen Ablauf:

1. Art des Sockels festlegen und erstellen
2. Binden des Sockets an eine Adresse und Port
3. Socket in den listen-Modus setzen und auf Verbindungen warten
4. Anfragen akzeptieren und Erstellen eines neuen Socket-Paares für Client
5. Datenaustausch über den Client-Sockel
6. Nach beenden der Kommunikation, den Client-Sockel wieder schließen

4 Durchführung

In diesem Kapitel wird die Vorgehensweise zum Erreichen der Aufgabenstellung beschrieben und auf die auftretenden Probleme eingegangen, sowie entsprechende Lösungsansätze erläutert.

Zunächst wurde sich grundlegend mit der Hardware und den Bibliotheken von ST vertraut gemacht und anschließend daran FreeRTOS integriert. Danach erfolgte die Implementation von lwIP und der SSL/TLS-Verschlüsselung. Zuletzt erfolgte die Verbindung mit einem MQTT-Broker.

4.1 Einarbeitung in die STM32-Hardware und HAL

Da der STM32F777 als Zielhardware zu Beginn dieser Arbeit, aufgrund von Lieferschwierigkeiten seitens ST, noch nicht zur Verfügung stand, wurde zunächst ein STM32F407 mit einem Cortex-M4-Prozessor verwendet. Dabei handelt es sich ebenfalls um eine SYSTEC-eigene STM32-basierte Referenzhardware. Im Gegensatz zum STM32F777 besitzt er keinen Crypto/Hash-Prozessor und weniger Speicher.

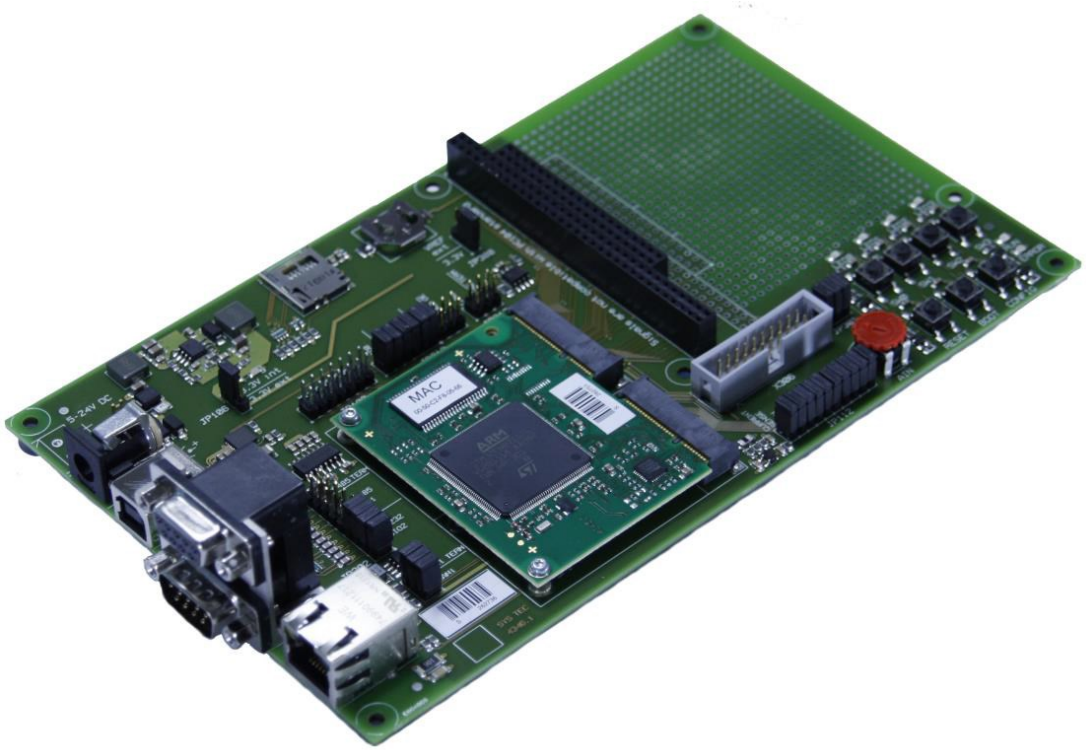

Abbildung 13 - PLCcore-F407 [33]

Für einen unkomplizierten Einstieg und um die Programmierung zu erleichtern, wurde die von ST erstellte und freigegebene HAL-Bibliothek verwendet. Der Begriff HAL bedeutet hardware abstraction layer. Im Gegensatz zu der Standardbibliothek werden in der HAL Funktionen und Abläufe, wie etwa Initialisierungen, zusammengefasst und dadurch die Programmierung vereinfacht. Beispielsweise bietet sie für die Benutzung von LEDs die Möglichkeit, eine Toggle-Funktion zu verwenden. Dadurch wird dem Benutzer die Arbeit abgenommen, den aktuellen Zustand der LED selbst abzufragen und ein entsprechendes Ein- oder Ausschalten auszuführen.

Um den Einstieg und die Benutzung der HAL weiterhin zu erleichtern, bietet ST ein eigenes Programmierinterface an: STM32CubeMX. Über ein graphisches Programmierinterface lassen sich sehr einfach Initialisierungen vornehmen und entsprechend Code generieren. Relevant wurde diese Form der Codeerzeugung nur für den Systemtakt (SystemCoreClock) des STM32F407. Ein Bild der Oberfläche ist in der folgenden Abbildung zu sehen. [34]

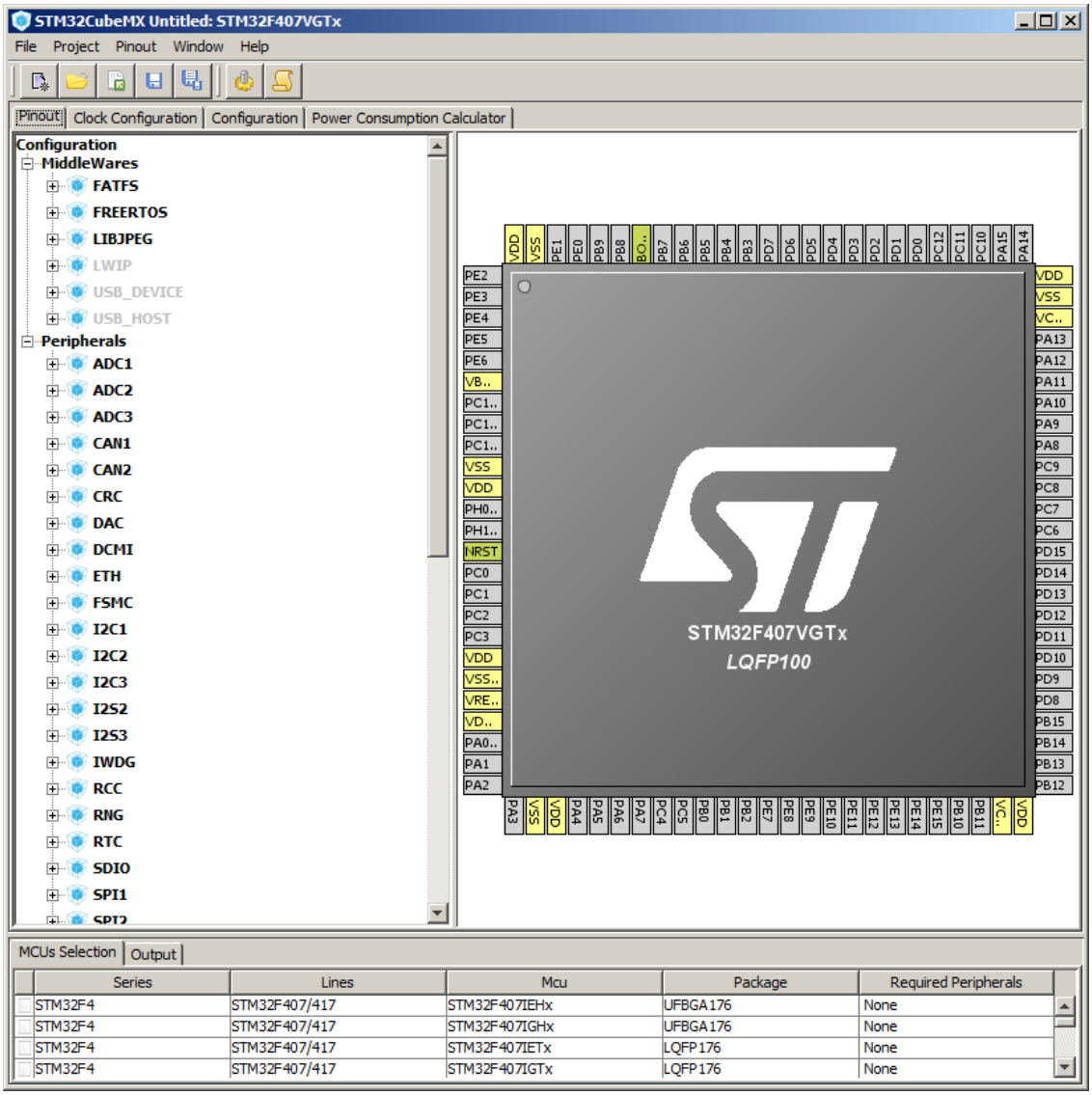

Abbildung 14 - Startansicht CubeMX

Die HAL-Bibliothek ist allerdings nicht fehlerfrei. Während der Einarbeitung und Erstellen einfacher Beispielprogramme stellte sich heraus, dass einige Interrupt-Handler nicht ausreichend oder nur als leere Funktionen vorhanden sind. Auch Prioritäten, allen voran die des SysTicks, sind nicht so, wie es erwartet wurde. Der SysTick dient als Zeitmessung für alle zeitbasierten Anwendungen, zum Beispiel für den Scheduler von FreeRTOS. Daher ist der SysTick eine der wichtigsten Funktionen und muss dementsprechend mit einer ausreichend hohen Priorität festgelegt werden. In der HAL wurde dieser aber standardmäßig mit der niedrigsten Priorität (0xF) eingestuft. Bei Cortex-M-Prozessoren ist eine hohe Zahl eine niedrige Priorität, so dass der SysTick deswegen unter Umständen von anderen Interrupts verdrängt werden könnte. Dadurch werden die Zeitintervalle nicht mehr eingehalten. Eine entsprechende Anpassung (auf 0x0, maximale Priorität) musste daher vorgenommen werden, um einen reibungslosen Ablauf zu garantieren.

Weiterhin stellt die Firma ARM, der Entwickler der Cortex-M-Prozessoren, eine CMSIS-Bibliothek zur Verfügung, die ebenfalls dem Programmierer die Arbeit erleichtert. CMSIS kümmert sich dabei beispielsweise um die Initialisierung des Systems, das Erstellen der Interruptvektortabelle und auch um Funktionen für die Benutzung eines Echtzeitbetriebssystems.

4.2 Implementierung von FreeRTOS

Für die Nutzung von FreeRTOS ist es notwendig, wichtige Konfigurationen vorzunehmen, sowie die bereits erwähnte HAL-Initialisierung durchzuführen. Standardmäßig sind bereits grundlegende Konfigurationen vorhanden. Für den optimalen und effizienteren Einsatz müssen diese angepasst werden. In der folgenden Abbildung werden einige Beispielkonfigurationen gezeigt.

```
#define configUSE_PREEMPTION            1
#define configUSE_IDLE_HOOK             0
#define configUSE_TICK_HOOK             0
#define configCPU_CLOCK_HZ              ( SystemCoreClock )
#define configTICK_RATE_HZ              ( ( TickType_t ) 1000 )
#define configMAX_PRIORITIES            ( 7 )
#define configMINIMAL_STACK_SIZE        ( ( uint16_t ) 128 )
#define configTOTAL_HEAP_SIZE           ( ( size_t ) ( 14 * 1024 ) )
#define configMAX_TASK_NAME_LEN         ( 16 )
#define configUSE_TRACE_FACILITY        1
#define configUSE_16_BIT_TICKS          0
#define configIDLE_SHOULD_YIELD         1
#define configUSE_MUTEXES               1
#define configQUEUE_REGISTRY_SIZE       8
#define configCHECK_FOR_STACK_OVERFLOW  0
#define configUSE_RECURSIVE_MUTEXES     1
#define configUSE_MALLOC_FAILED_HOOK    0
#define configUSE_APPLICATION_TASK_TAG  0
#define configUSE_COUNTING_SEMAPHORES   1
#define configGENERATE_RUN_TIME_STATS   0
```

Abbildung 15 - Beispielkonfiguration FreeRTOS

Weiterhin müssen vor der Benutzung von FreeRTOS dringend benötigte Initialisierungen vorgenommen werden.

4.2.1 Der Systick-Interrupt

Damit der Scheduler ordnungsgemäß zwischen den Tasks hin und her wechseln kann, benötigt man einen Zeitgeber oder Zähler. Im Fall des Cortex-M gibt es dafür den bereits erwähnten SysTick-Interrupt. Dieser wird bei der Initialisierung der HAL konfiguriert und aktiviert. Standardmäßig löst der SysTick jede Millisekunde aus, kann aber nach Wunsch selbst angepasst werden. In der nachfolgenden Abbildung ist die von der HAL durchgeführte Konfiguration zu sehen.

```
__weak HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority)
{
  /*Configure the SysTick to have interrupt in 1ms time basis*/
  HAL_SYSTICK_Config(SystemCoreClock/1000U);

  /*Configure the SysTick IRQ priority */
  HAL_NVIC_SetPriority(SysTick_IRQn, TickPriority ,0U);

  /* Return function status */
  return HAL_OK;
}
```

Abbildung 16 - Konfiguration des Systicks

4.2.2 Priorisierung der Interruptbits

Weiterhin ist es von größter Wichtigkeit, dass alle Interrupt-Bits präventive (preemt) Priorität zugewiesen bekommen, bevor das RTOS startet. Die Prioritäten sind in einem 8-Bit-Register festgelegt und in zwei Gruppen unterteilt: präventive Priorität und Sub-Priorität. Präventive Priorität definiert, ob ein Interrupt einen anderen mit gleicher (präventiver) Priorität unterbrechen kann. Die Sub-priorität entscheidet, welcher Interrupt Vorrang hat, wenn beide Interrupts dieselbe präventive Priorität haben. Diese Konfiguration wird bei der Initialisierung der HAL durchgeführt. [35]

```
/* Set Interrupt Group Priority */
HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4);
```

Abbildung 17 - präventive Interruptpriorität

4.2.3 Speicherverwaltung

Für die Speicherverwaltung gibt es in FreeRTOS im Wesentlichen zwei Möglichkeiten der Anpassung. Zum einen lässt sich die Art des Speicherallokierens zwischen dynamisch und statisch wechseln, zum anderen ändern sich die Funktionen für allokieren bzw. freigeben von Speicher, je nach verwendeter heap-Implementierung.

4.2.3.1 *Dynamische und statische Speicherallokation*

Die dynamische Speicherallokation befreit den Programmierer davon, sich selbst Gedanken darüber zu machen, wie der Speicher, zum Beispiel für benötigte Tasks, zu verwalten ist. Funktionen benötigen demnach unter anderem weniger Parameter, die Speicherallokation erfolgt automatisch über RTOS-Funktionen und benutzter RAM kann wieder freigegeben und für andere Aktionen verwendet werden.

Im Gegensatz dazu, gibt die statische Speicherallokation dem Programmierer mehr Kontrolle. So kann selbst festgelegt werden, wo genau RTOS-Objekte im Speicher platziert werden. Ebenso muss sich nicht mit auftretenden Fehlern, bei der automatischen Speicherallokation beschäftigt werden. Allerdings setzt diese Version natürlich auch ein höheres Verständnis der RTOS-Funktionen voraus. [36]

Aufgrund dessen, dass der Großteil automatisch vom RTOS übernommen wird, einige Funktionsparameter entfallen und die Auswahl der Implementierungen (siehe 5.2.3.2) höher ist, wurde sich dazu entschieden, die dynamische Speicherallokation zu verwenden.

4.2.3.2 *Implementierung der Speicherallokation*

Weiterhin besteht die Möglichkeit, sich zwischen verschiedenen Implementationen der Speicherallokation zu entscheiden. Zum Zeitpunkt dieser Arbeit gibt es fünf verschiedene Varianten. Die Implementationen sind in getrennte C-Dateien aufgeteilt, heap_1, heap_2, usw. In der folgenden Auflistung wird auf die jeweilige Variante und deren Vor- oder Nachteile eingegangen.

heap_1 Sie ist die simpelste Implementierung und erlaubt es nicht, allokierten Speicher freizugeben und damit wieder zu verwenden. Dabei handelt es sich um die einzige Variante, die mit statischer Speicherallokation benutzt werden kann.

heap_2 Die Freigabe von Speicher ist im Gegensatz zu Variante 1 möglich. Freie benachbarte Blöcke werden allerdings nicht in einen großen Block kombiniert.

heap_3 Wie bei heap_2 ist die Freigabe von allokiertem Speicher möglich, allerdings handelt es sich um einen simplen Wrapper der die herkömmlichen malloc() und free() C-Funktionen einsetzt und diese Tasksicher macht. Das Verhalten ist nicht deterministisch.

heap_4 Um Fragmentierung zu vermeiden, verbindet Variante 4 benachbarte freie Speicherblöcke. Sie beinhaltet außerdem die Option für absolute Adressierung.

heap_5 Sie ist wie Variante 4, ermöglicht es aber heap-Speicher über mehrere nicht-zusammenhängende Speicherbereiche zu legen.

Damit bietet FreeRTOS ein umfangreiches Angebot, die Speicherverwaltung seinen Bedürfnissen anzupassen. Weiterhin steht es einem jederzeit frei, diese Implementationen zu verändern oder eine eigene Variante zu programmieren. [37]

Es ist möglich, jede der Implementierungsarten zu verwenden. Da die dynamische Speicherallokation gewählt wurde (siehe 5.2.3.1), kann allerdings in dieser Arbeit heap_1 nicht verwendet werden. Die Varianten 2 und 3 schienen zunächst wegen ihrer Simplizität ansprechend, allerdings wiegen diese möglichen Vorteile den Nachteil, des nicht-deterministischen Verhaltens, nicht auf. Wenn sich das System zu jedem beliebigen Zeitpunkt nicht-deterministisch verhalten kann, macht dies die Fehlersuche und das Debuggen unnötig schwer und kompliziert. Auch das nicht Zusammenfassen benachbarter freier Speicherblöcke könnte langfristig zu einem Problem werden. Es wurde sich daher für die Verwendung von heap_5 entschieden, auch wenn diese Variante sehr umfangreich ist.

4.2.4 Taskerzeugung

Damit nun FreeRTOS verwendet werden kann, müssen noch Tasks erstellt werden. Je nach dem, für welche Art des Speicherallokierens man sich entschieden hat, sind unterschiedliche Funktionen für die Taskerzeugung definiert. Auf eine Erklärung für das statische Speicherallokieren wird verzichtet, da ausschließlich die dynamische Variante verwendet wurde. Die dazugehörige Deklaration der Taskerzeugungsfunktion ist in der nachfolgenden Abbildung zu sehen.

```
BaseType_t xTaskCreate( TaskFunction_t pxTaskCode,
                        const char * const pcName,
                        const uint16_t usStackDepth,
                        void * const pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t * const pxCreatedTask )
```

Abbildung 18 - Funktion zur Erzeugung eines Task

Der Task wird dadurch erstellt und der Liste bereits laufender Tasks hinzugefügt. Weiterhin wird von der Speicherverwaltung der benötigte Speicher, für den Zustand in dem er sich befindet sowie dessen Stack zugewiesen. [41]

pxTaskCode Ein Zeiger zu der eigentlichen Taskfunktion. Tasks sind in der Regel als Endlosschleife zu implementieren, allerdings können und müssen Tasks sich selber löschen, wenn sie nicht mehr gebraucht werden.

pcName	Der Name des Tasks dient hauptsächlich zum Debuggen, kann aber auch dazu benutzt werden, einen Task Handle zu erzeugen.
usStackDepth	Größe der zu allokierenden Stackgröße in Wörter (nicht Bytes).
pvParameters	Ein Parameter, der dem Task übergeben werden soll. Wenn eine Adresse einer Variable übergeben wird, muss diese Variable existieren, wenn der erzeugte Task ausgeführt wird.
uxPrioritiy	Die Priorität, mit der der Task ausgeführt werden soll
pxCreatedTask	Übergibt einen Handler an den erzeugten Task. Ist optional und kann NULL gesetzt werden.

Ebenso wird mit der Verwendung von CMSIS eine weitere Möglichkeit für die Erzeugung von Tasks geboten. Anders als bei der Standardvariante des FreeRTOS, wird zunächst ein Task über eine Struktur definiert. Erst mit einer CMSIS-Funktion wird dann der definierte Task erzeugt, in dem unter anderem auch die bereits erwähnte xTaskCreate-Funktion aufgerufen wird. Die Struktur und Funktion sind in den folgenden Abbildungen zu sehen.

```
typedef struct os_thread_def  {
  char                    *name;
  os_pthread              pthread;
  osPriority              tpriority;
  uint32_t                instances;
  uint32_t                stacksize;
#if( configSUPPORT_STATIC_ALLOCATION == 1 )
  uint32_t                *buffer;
  osStaticThreadDef_t     *controlblock;
#endif
} osThreadDef_t;
```

Abbildung 19 - Definition Task CMSIS

```
osThreadId osThreadCreate (const osThreadDef_t *thread_def, void *argument)
{
  TaskHandle_t handle;

#if( configSUPPORT_STATIC_ALLOCATION == 1 ) &&  ( configSUPPORT_DYNAMIC_ALLOCATION == 1 )
  if((thread_def->buffer != NULL) && (thread_def->controlblock != NULL)) {
    handle = xTaskCreateStatic((TaskFunction_t)thread_def->pthread,(const portCHAR *)thread_def->name,
            thread_def->stacksize, argument, makeFreeRtosPriority(thread_def->tpriority),
            thread_def->buffer, thread_def->controlblock);
  }
  else {
    if (xTaskCreate((TaskFunction_t)thread_def->pthread,(const portCHAR *)thread_def->name,
            thread_def->stacksize, argument, makeFreeRtosPriority(thread_def->tpriority),
            &handle) != pdPASS)  {
      return NULL;
    }
  }
#elif( configSUPPORT_STATIC_ALLOCATION == 1 )

    handle = xTaskCreateStatic((TaskFunction_t)thread_def->pthread,(const portCHAR *)thread_def->name,
            thread_def->stacksize, argument, makeFreeRtosPriority(thread_def->tpriority),
            thread_def->buffer, thread_def->controlblock);
#else
  if (xTaskCreate((TaskFunction_t)thread_def->pthread,(const portCHAR *)thread_def->name,
            thread_def->stacksize, argument, makeFreeRtosPriority(thread_def->tpriority),
            &handle) != pdPASS)  {
    return NULL;
  }
#endif

  return handle;
}
```

Abbildung 20 - Erzeugung Task CMSIS

Die CMSIS-Variante bietet dem Anwender den Vorteil, dass überprüft wird, welche Art des Speicherallokierens definiert wurde und anhand dessen die entsprechende Taskerzeugungsfunktion benutzt wird. Deshalb wurde, obwohl nur die Methode des dynamischen Speicherallokierens gewählt wurde, sich dazu entschieden, die CMSIS-Variante der Taskerzeugung zu verwenden. Denn ohne dass der eigentliche Quellcode geändert werden müsste, könnte das resultierende Programm so auch auf anderer Hardware und mit anderen Konfigurationen des RTOS verwendet werden.

4.2.5 Anwendungsbeispiel

Um die Funktionalität und das Wechseln der Tasks zu testen, wurde eine einfache Beispielanwendung geschrieben. Dabei werden in verschiedenen Zeitintervallen und in je separaten Tasks, die LED 1 und 2 des STM32F407 zum Blinken gebracht. Die Taskerzeugungsfunktionen und jeweiligen Tasks sind in den folgenden zwei Abbildungen zu sehen.

```
osThreadDef(Test, OS_TEST, osPriorityHigh, 0, configMINIMAL_STACK_SIZE);
osThreadCreate (osThread(Test), NULL);

osThreadDef(Test_2, OS_TEST_2, osPriorityHigh, 0, configMINIMAL_STACK_SIZE);
osThreadCreate (osThread(Test_2), NULL);
```

Abbildung 21 - Taskerzeugung

```c
static void OS_TEST(void)
{
    while(1)
    {
        BSP_LED_Toggle(LED1);
        osDelay(200);
    }
}

static void OS_TEST_2(void)
{
    while(1)
    {
        BSP_LED_Toggle(LED2);
        osDelay(500);
    }
}
```

Abbildung 22 - LED-Tasks

Wie bereits erklärt, werden die CMSIS- und HAL-Funktionen verwendet. Die Tasks befinden sich je in einer Endlosschleife und wechseln den momentanen Zustand der LED in einem Zeitintervall von 200 bzw. 500 Millisekunden.

4.3 Einbindung von lwIP

Nach erfolgreicher Implementierung von FreeRTOS, wurde im nächsten Schritt lwIP eingebunden. Analog zu FreeRTOS müssen Konfigurationen vorgenommen. Dabei gilt es unter anderem zu entscheiden, ob lwIP mit einem Betriebssystem läuft, welche APIs und ob DHCP oder andere Protokolle benutzt werden sollen.

4.3.1 Programmierschnittstellen

LwIP stellt im Wesentlichen drei verschiedene APIs zur Verfügung. Low-level core/callback oder raw API und zwei höhere sequenzielle APIs, netconn und socket.

raw Sie ist ereignisorientiert und dazu gedacht, ohne ein Betriebssystem benutzt zu werden. Daher ist sie für diese Arbeit von keiner Bedeutung. Allerdings bietet sie die beste Performance. [38]

callback Ähnlich funktionierend wie die raw-API, ist die callback-API allerdings für den Betrieb mit einem RTOS konzipiert. Sie kann mit netconn und socket benutzt werden. [38]

netconn Für die Nutzung ist ein Betriebssystem erforderlich, da netconn auf Tasks angewiesen ist. Die Be- und Verarbeitung von Paketen, sowohl bei In- als auch Output, werden in einem zugewiesenen Task durchgeführt. [39]

socket Ebenso wie netconn, benötigt diese API ein Betriebssystem. Sie erlaubt es Sockets zu erzeugen, die mit den Posix- und BSD-Sockets kompatibel sind. [40]

4.3.2 Initialisierung der Hardware

Damit die Hardware für die Netzwerkkommunikation benutzt werden kann, muss zuerst die PHY initialisiert werden. Eine PHY ist ein integrierter Schaltkreis, der für die Kodierung und Dekodierung von Signalen zwischen einem digitalen und modulierten analogen System dient. Sie umfasst den Ethernetanschluss und die dazugehörigen Steuereinheiten. Weiterhin ist PHY eine gebräuchliche Abkürzung für die Bitübertragungsschicht (physical layer) des OSI-Modells (siehe Kapitel 3.2.1). [42, S. 495]

ST bietet in seiner HAL-Bibliothek bereits eine Beispielkonfiguration für ihre selbst vertriebene Development-Hardware an. Allerdings sind diese, durch andere GPIO-PIN-Belegung, nicht auf der SYS TEC-eigenen Hardware anwendbar. Daher wurde eine entsprechende Anpassung in der Headerdatei „stm32f4xx_hal_conf.h“ durchgeführt. Standardmäßig wird in der HAL eine PHY vom Typ DP83848 verwendet, benutzt wurde aber eine PHY vom Typ KSZ8051RNL. Folgend sind die vorgenommenen Änderungen abgebildet. Auf der linken Seite ist die PIN-Belegung der PHY vom Typ DP83848 zu sehen, rechts der PHY vom Typ KSZ8051RNL.

```
ETH_MDIO -------------------------> PA2
ETH_MDC --------------------------> PC1
ETH_PPS_OUT ----------------------> PB5
ETH_MII_CRS ----------------------> PH2
ETH_MII_COL ----------------------> PH3
ETH_MII_RX_ER --------------------> PI10
ETH_MII_RXD2 ---------------------> PH6
ETH_MII_RXD3 ---------------------> PH7
ETH_MII_TX_CLK -------------------> PC3
ETH_MII_TXD2 ---------------------> PC2
ETH_MII_TXD3 ---------------------> PB8
ETH_MII_RX_CLK/ETH_RMII_REF_CLK---> PA1
ETH_MII_RX_DV/ETH_RMII_CRS_DV ----> PA7
ETH_MII_RXD0/ETH_RMII_RXD0 -------> PC4
ETH_MII_RXD1/ETH_RMII_RXD1 -------> PC5
ETH_MII_TX_EN/ETH_RMII_TX_EN -----> PG11
ETH_MII_TXD0/ETH_RMII_TXD0 -------> PG13
ETH_MII_TXD1/ETH_RMII_TXD1 -------> PG14
```

```
ETH_MII_RX_CLK/ETH_RMII_REF_CLK---> PA1
ETH_MDIO -------------------------> PA2
ETH_MII_RX_DV/ETH_RMII_CRS_DV ----> PA7
ETH_MII_TX_EN/ETH_RMII_TX_EN -----> PB11
ETH_MDC --------------------------> PC1
ETH_MII_RXD0/ETH_RMII_RXD0 -------> PC4
ETH_MII_RXD1/ETH_RMII_RXD1 -------> PC5
ETH_MII_TXD0/ETH_RMII_TXD0 -------> PG13
ETH_MII_TXD1/ETH_RMII_TXD1 -------> PG14
```

Abbildung 23 - PHY GPIO

Da die PHY nicht nur anders angeschlossen, sondern auch von einem anderen Typ ist, mussten nicht nur die GPIOs angepasst werden, sondern ebenso die entsprechend verwendeten Register. In den beiden folgenden Abbildungen ist auszugsweise dargestellt,

welche Änderungen an den Registereinträgen vorgenommen wurden. Beispielsweise sind die Register MICR und MISR beim KSZ8051RNL zu einem gemeinsam benutzten Register zusammengefasst. Für die auftretenden Interrupts gibt es außerdem, im Gegensatz zum DP83848**,** kein gemeinsames Register, sondern sind diese jeweils nach Art des auftretenden Interrupts getrennt.

```
/* Section 3: Common PHY Registers */

#define PHY_BCR                          ((uint16_t)0x0000)
#define PHY_BSR                          ((uint16_t)0x0001)

#define PHY_RESET                        ((uint16_t)0x8000)
#define PHY_LOOPBACK                     ((uint16_t)0x4000)
#define PHY_FULLDUPLEX_100M              ((uint16_t)0x2100)
#define PHY_AUTONEGOTIATION              ((uint16_t)0x1000)


#define PHY_LINKED_STATUS                ((uint16_t)0x0004)
#define PHY_AUTONEGO_COMPLETE            ((uint16_t)0x0020U)

/* Section 4: Extended PHY Registers */
#define PHY_SR                           ((uint16_t)0x0010U)

#define PHY_DUPLEX_STATUS                ((uint16_t)0x0004)
#define PHY_SPEED_STATUS                 ((uint16_t)0x0020)
```

Abbildung 24 - Register PHY KSZ8051RNL

```
/* Section 3: Common PHY Registers */

#define PHY_BCR                          ((uint16_t)0x00)
#define PHY_BSR                          ((uint16_t)0x01)

#define PHY_RESET                        ((uint16_t)0x8000)
#define PHY_LOOPBACK                     ((uint16_t)0x4000)
#define PHY_FULLDUPLEX_100M              ((uint16_t)0x2100)
#define PHY_HALFDUPLEX_100M              ((uint16_t)0x2000)
#define PHY_FULLDUPLEX_10M               ((uint16_t)0x0100)
#define PHY_HALFDUPLEX_10M               ((uint16_t)0x0000)
#define PHY_AUTONEGOTIATION              ((uint16_t)0x1000)
#define PHY_RESTART_AUTONEGOTIATION      ((uint16_t)0x0200)
#define PHY_POWERDOWN                    ((uint16_t)0x0800)
#define PHY_ISOLATE                      ((uint16_t)0x0400)

#define PHY_AUTONEGO_COMPLETE            ((uint16_t)0x0020)
#define PHY_LINKED_STATUS                ((uint16_t)0x0004)
#define PHY_JABBER_DETECTION             ((uint16_t)0x0002)

/* Section 4: Extended PHY Registers */

#define PHY_SR                           ((uint16_t)0x10)
#define PHY_MICR                         ((uint16_t)0x11)
#define PHY_MISR                         ((uint16_t)0x12)

#define PHY_LINK_STATUS                  ((uint16_t)0x0001)
#define PHY_SPEED_STATUS                 ((uint16_t)0x0002)
#define PHY_DUPLEX_STATUS                ((uint16_t)0x0004)
```

Abbildung 25 - Register PHY DP83848

4.3.3 Hinzufügen eines Netzwerkinterface

Zur Verwendung von lwIP, muss weiterhin ein entsprechendes Netzwerkinterface hinzugefügt werden. In der folgenden Abbildung ist die dazugehörige Funktion zu sehen.

```
/* add the network interface */
netif_add(&netif, &addr, &netmask, &gw, NULL, &ethernetif_init, &ethernet_input);
          6       1      2        3           4                  5
```

Abbildung 26 - Hinzufügen eines Netzwerkinterfaces

Die Funktion netif_add erstellt das Netzwerkinterface mit der IP-Adresse des Mikrocontrollers (1), Netzmaske (2), Gateway (3), übergibt gleichzeitig eine Initialisierungs- (4) und Input-Funktion (5) und speichert dies in einem neuen Netzwerkinterface (6) ab.

Die übergebene Input-Funktion wird dabei durch einen Task aufgerufen, der Pakete aus dem erzeugtem Netzwerkinterface ausliest und selbst durch einen Semaphor aktiviert wird, der vom Empfangsinterrupt freigegeben wird.

Die Funktionsweise läuft dabei in folgender Reihenfolge ab:

1. Empfangsinterrupt registriert
2. Freigabe eines binären Semaphors
3. Auf Semaphor wartender Task wird freigegeben
4. Task liest das empfangene Paket aus
5. Paket wird zur weiteren Verarbeitung an die Input-Funktion übergeben
6. Task begibt sich wieder in Warteposition

4.3.4 Testverbindung und Echo

Um eine funktionierende Verbindung mit lwIP zu testen und in kontrolliertem Rahmen zu debuggen, wurde eine Testanwendung geschrieben, bei der über getrennte Tasks, je ein TCP- und UDP-Socket, ein Echo an den Sender zurückgeschickt wurde. Dafür wurde der von Microsoft bereitgestellten Telnet-Client für TCP benutzt und ein Sendetool für UDP, geschrieben von Herrn Dipl.-Ing. Ronald Sieber.

Zusätzlich zu der umgesetzten Echo-Funktion, wird im UDP-Thread der Inhalt der empfangenen Nachricht überprüft. Sollte die Nachricht „LED ON“ oder „LED OFF“ sein, wird entsprechend die LED 1 des Systems ein- bzw. ausgeschaltet. Ebenso wird in dem Fall kein Echo gesendet, sondern die entsprechende Nachricht „ON“ oder „OFF“. Folgend ist das Echo des UDP-Threads zu sehen.

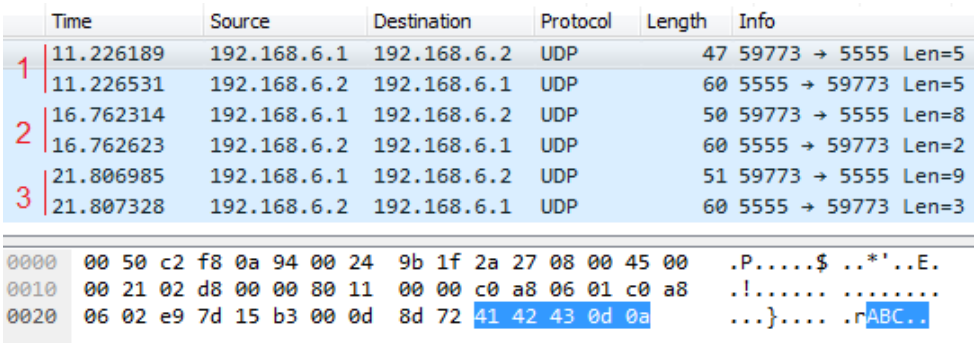

Abbildung 27 - UDP-Echo

Es werden drei Nachrichten geschickt. Bei der ersten (1) und im unteren Fenster gezeigten Nachricht handelt es sich um die Buchstaben „ABC“, die zweite Nachricht (2) ist der „LED ON“-Befehl und die dritte Nachricht (3) der „LED OFF“-Befehl.

4.4 Verschlüsselung mit SSL/TLS

LwIP bietet keine eigene Möglichkeit zur Verschlüsselung. Daher ist es notwendig, eine zusätzliche Funktionsbibliothek heranzuziehen und zu implementieren. Nach umfangreicher Recherche standen im wesentlichen drei Möglichkeiten zur engeren Auswahl: OpenSSL, mbedTLS und WolfSSL.

4.4.1 Vergleich der Funktionsbibliotheken

OpenSSL ist ein Open-Source-Projekt und kann auch für jede kommerzielle Anwendung benutzt werden. Es ist eine der am häufigsten verwendete SSL-Bibliotheken. Eine vollständig kompilierte Bibliothek, benötigt allerdings mit circa 800 KB viel Speicher und kann damit auf vielen eingebetteten System nicht eingesetzt werden.

MbedTLS war früher unter dem Namen PolarSSL bekannt und steht in direkter Konkurrenz zu OpenSSL. Es ist ohne Abhängigkeiten vollständig in der Programmiersprache C geschrieben. Dadurch ist es auf vielen Betriebssystemen (z.B. Linux, Windows, Android, FreeRTOS) und verschiedenen Prozessorarchitekturen (z.B. ARM, x86, MIPS) einsetzbar. Dafür, dass es nicht so weit verbreitet ist, wie OpenSSL, ist es sehr umfassend dokumentiert, die Funktionsweise nachvollziehbar strukturiert und der verwendete Speicher sehr gering. Ebenso bietet ST Implementationsbeispiele an, so dass ein Einstieg, gerade für die Anwendung mit STM32-Systemen, unkompliziert ist. Allerdings sind die umfangreichen Konfigurationsmöglichkeiten für Einsteiger unter Umständen überfordernd. Weiterhin ist mbedTLS mittlerweile offiziell Teil von ARM. [29] [30]

Ähnlichen Möglichkeiten bietet auch WolfSSL, ehemals CyaSSL. Jedoch ist die Funktionsweise undurchsichtiger und schwieriger nachzuvollziehen. Hinzu kommt, dass es nicht

so umfangreich dokumentiert ist und weniger Anwendungsbeispiele, wie etwa bei OpenSSL oder mbedTLS, existieren.

Durch den erleichterten Einstieg, die bessere Dokumentation, den geringeren Speicherverbrauch und den direkten Support von ARM-Prozessoren, fiel die Entscheidung letztlich zur Verwendung von mbedTLS.

4.4.2 Verwendung und Ablauf von mbedTLS

Dieses Kapitel geht ausführlich auf die Struktur und den Ablauf des TLS-Verbindungsaufbaus mit mbedTLS ein. Zunächst müssen bei der Konfiguration verwendete Module und Quellen für die Verschlüsselung bzw. Entropie ausgewählt werden. In der untenstehenden Abbildung sind beispielhaft einige Konfigurationsmöglichkeiten aufgezeigt.

```
/* mbed TLS modules */
#define MBEDTLS_AES_C
#define MBEDTLS_ASN1_PARSE_C
#define MBEDTLS_ASN1_WRITE_C
#define MBEDTLS_BIGNUM_C
#define MBEDTLS_CIPHER_C
#define MBEDTLS_CTR_DRBG_C
#define MBEDTLS_ECDH_C
#define MBEDTLS_ECDSA_C
#define MBEDTLS_ECP_C
#define MBEDTLS ENTROPY C

/* For test certificates */
#define MBEDTLS_BASE64_C
#define MBEDTLS_CERTS_C
#define MBEDTLS_PEM_PARSE_C
```

Abbildung 28 - Auszug Konfiguration mbedTLS

Weiterhin bietet mbedTLS bereits vorgefertigte Schlüssel- und Entropiequellen an. Da die Beispielschlüssel- und Zertifikate aber jedem zugänglich sind und heruntergeladen werden können, ist deren Sicherheit nicht für kommerzielle Zwecke ausreichend. Um einen optimalen Schutz garantieren zu können, sollten daher eigene Schlüssel- und Entropie-Implementationen vorgenommen werden. Die vorgefertigten Schlüssel und Zertifikate sind allerdings ausreichend für Testapplikationen. Sie liegen in Form von Zeichenketten bzw. -blöcken in der Datei „certs.c“ vor und werden wie der Rest der Bibliothek kompiliert. Für das Verwenden eigener Zertifikate und Schlüssel, müssen die Zeichenblöcke in dieser Datei entsprechend geändert werden. Folgende Abbildung zeigt, das Beispiel Zertifikat für einen mbedTLS-Client.

```
"-----BEGIN CERTIFICATE-----\r\n"
"MIICLDCCAbKgAwIBAgIBDTAKBggqhkjOPQQDAjA+MQswCQYDVQQGEwJOTDERMA8G\r\n"
"A1UEChMIUG9sYXJTU0wxHDAaBgNVBAMTE1BvbGFyc3NsIFR1c3QgRUMgQ0EwHhcN\r\n"
"MTMwOTI0MTU1MjA0WhcNMjMwOTIyMTU1MjA0WjBBMQswCQYDVQQGEwJOTDERMA8G\r\n"
"A1UEChMIUG9sYXJTU0wxHzAdBgNVBAMTFlBvbGFyU1NMIFR1c3QgQ2xpZW50IDIw\r\n"
"WTATBgcqhkjOPQIBBggqhkjOPQMBBwNCAARX5a6xc9/TrLuTuIH/Eq7u5lOszlVT\r\n"
"9jQOzC7jYyUL35ji81xgNpbA1RgUcOV/n9VLRRjlsGzVXPiWj4dwo+THo4GdMIGa\r\n"
"MAkGA1UdEwQCMAAwHQYDVR0OBBYEFHoAX4Zk/OBd5REQO7LmO8QmP8/iMG4GA1Ud\r\n"
"IwRnMGWAFJ1tICRJAT8ry3i1Gbx+JMnb+zZ8oUKkQDA+MQswCQYDVQQGEwJOTDER\r\n"
"MA8GA1UEChMIUG9sYXJTU0wxHDAaBgNVBAMTE1BvbGFyc3NsIFR1c3QgRUMgQ0GC\r\n"
"CQDBQ+J+YkPM6DAKBggqhkjOPQQDAgNoADBlAjBKZQ17IIOimbmoD/yN7o89u3BM\r\n"
"lgOsjnhw3fIOoLIWy2WOGsk/LGF++DzvrRzuNiACMQCd8iem1XS4JK7haj8xocpU\r\n"
"LwjQje5PDGHfd3h9tP38Qknu5bJqws0md2KOKHyeV0U=\r\n"
"-----END CERTIFICATE-----\r\n";
```

Abbildung 29 - Client-Zertifikat

Entropie ist in der Informationstheorie Maß für den mittleren Informationsgehalt einer Nachricht. In einem Computersystem ist es schwierig, echte Zufallszahlen zu erzeugen. Um die Qualität der Zufallszahlen zu erhöhen, werden schwer vorhersagbare Werte, wie zum Beispiel Mausbewegungen, Tastaturanschlag- oder Netzwerk-Timings, in einem Entropie-Pool gesammelt. Die in diesem Pool gesammelten Zahlen werden dann in einem (Pseudo-)Zufallsgenerator verwendet, um wiederrum bessere Zufallszahlen erzeugen zu können. Im Rahmen dieser Arbeit war es ausreichend, die vorhandenen Entropiequellen zu verwenden. [49]

Die einzelnen Schritte, die beim Aufbau einer verschlüsselten Verbindung mit mbedTLS vorgenommen werden müssen, sind in der nachfolgenden Abbildung aufgelistet und werden anschließend genauer erläutert.

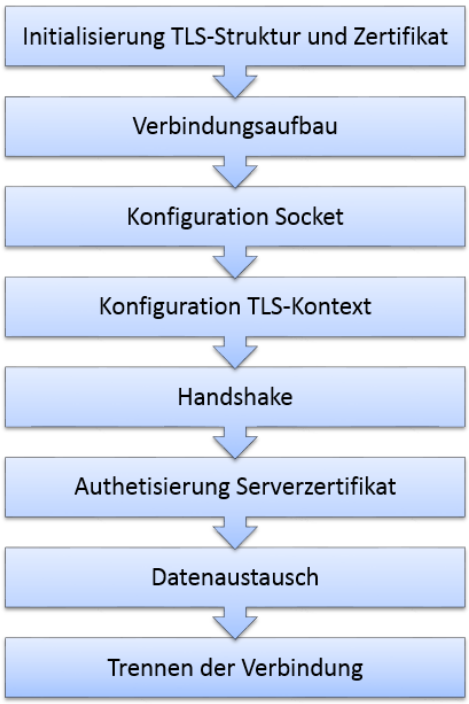

Abbildung 30 - Ablauf Verbindungsaufbau mbedTLS

Bevor eine TLS-Verbindung erstellt werden kann, müssen die TLS-Strukturen und RNG initialisiert werden, wie in den nachfolgenden Abbildungen zu sehen ist. Der Rückgabewert (ret) der meisten mbedTLS-Funktionen ist bei erfolgreicher Ausführung stets 0. Um daher Fehler abzufangen, befinden sich alle Funktionen innerhalb von if-Abfragen, die den Rückgabewert mit 0 vergleichen. Bei Ungleichheit wird eine Debug-Nachricht für den Programmierer erzeugt und der Verbindungsaufbau beendet.

```
mbedtls_net_init(&server_fd);
mbedtls_ssl_init(&ssl);
mbedtls_ssl_config_init(&conf);
mbedtls_x509_crt_init(&cacert);
mbedtls_ctr_drbg_init(&ctr_drbg);

mbedtls_printf( "\n  . Seeding the random number generator..." );

mbedtls_entropy_init( &entropy );
len = strlen((char *)pers);
if( ( ret = mbedtls_ctr_drbg_seed( &ctr_drbg, mbedtls_entropy_func, &entropy,
                                   (const unsigned char *) pers, len ) ) != 0 )
{
  mbedtls_printf( " failed\n  ! mbedtls_ctr_drbg_seed returned %d\n", ret );
  goto exit;
}
```

Abbildung 31 - Initialisierung TLS-Strukturen

Für die spätere Verwendung müssen ebenso vorhandene Zertifikate und Schlüssel initialisiert und geladen werden, wie in der nachfolgenden Abbildung dargestellt ist.

```
/*
 * 0. Initialize certificates
 */
mbedtls_printf( "  . Loading the CA root certificate ..." );

ret = mbedtls_x509_crt_parse( &cacert, (const unsigned char *) mbedtls_test_cas_pem,
                              mbedtls_test_cas_pem_len );
if( ret < 0 )
{
  mbedtls_printf( " failed\n  !  mbedtls_x509_crt_parse returned -0x%x\n\n", -ret );
  goto exit;
}
```

Abbildung 32 - Initialisierung Zertifikat

Nach diesen Schritten kann der reguläre Verbindungsaufbau, die Funktion dafür ist in der folgenden Abbildung zu sehen, durchgeführt werden. Dieser ist ähnlich wie bei einer regulären TCP-Verbindung aufgebaut und verwendet eine abgewandelte und dem TLS-Vorgang angepasste Socket-API.

```
if( ( ret = mbedtls_net_connect( &server_fd, SERVER_NAME,
                                 SERVER_PORT, MBEDTLS_NET_PROTO_TCP ) ) != 0 )
{
  mbedtls_printf( " failed\n  ! mbedtls_net_connect returned %d\n\n", ret );
  goto exit;
}

mbedtls_printf( " ok\n" );
```

Abbildung 33 - Verbindungsaufbau mbedTLS

Nun werden die Endpunktart (Client oder Server), Transportart (Datagram oder Stream) und Sicherheitsmerkmale des Sockets festgelegt. Ein Beispiel für einen Stream-Client-Socket ist in der nachfolgenden Abbildung zu sehen.

```
if( ( ret = mbedtls_ssl_config_defaults( &conf,
                MBEDTLS_SSL_IS_CLIENT,
                MBEDTLS_SSL_TRANSPORT_STREAM,
                MBEDTLS_SSL_PRESET_DEFAULT ) ) != 0 )
{
  mbedtls_printf( " failed\n  ! mbedtls_ssl_config_defaults returned %d\n\n", ret );
  goto exit;
}
```

Abbildung 34 - TLS Socket-Konfiguration

Im nächsten Schritt, werden abschließende Konfigurationen für den verwendeten TLS-Kontext vorgenommen und die verwendeten Input- und Output-Funktionen übergeben. Dieser Ablauf ist in der folgenden Abbildung dargestellt.

```
mbedtls_ssl_conf_authmode( &conf, MBEDTLS_SSL_VERIFY_OPTIONAL );
mbedtls_ssl_conf_rng( &conf, mbedtls_ctr_drbg_random, &ctr_drbg );
mbedtls_ssl_conf_ca_chain( &conf, &cacert, NULL );

if( ( ret = mbedtls_ssl_setup( &ssl, &conf ) ) != 0 )
{
  mbedtls_printf( " failed\n  ! mbedtls_ssl_setup returned %d\n\n", ret );
  goto exit;
}

if( ( ret = mbedtls_ssl_set_hostname( &ssl, "mbed TLS Server 1" ) ) != 0 )
{
  mbedtls_printf( " failed\n  ! mbedtls_ssl_set_hostname returned %d\n\n", ret );
  goto exit;
}

mbedtls_ssl_set_bio( &ssl, &server_fd, mbedtls_net_send, mbedtls_net_recv, NULL );
```

Abbildung 35 - Abschließende Konfiguration des TLS-Kontext

Bis hier ist der Client mit dem Server verbunden, eine verschlüsselte Kommunikation aber noch nicht möglich. Im nächsten Schritt, dem Handshake, sendet der Client dem Server eine Liste an Verschlüsselungsarten, die er unterstützt. Der Server wählt sich daraus die stärkste Variante aus und sendet sein Zertifikat und den Schlüssel. Darauf antwortet der Client mit seinem eigenen Schlüssel, optional auch mit einem Zertifikat. Untenstehende Abbildung zeigt die benutzte Funktion.

```
mbedtls_printf( "  . Performing the SSL/TLS handshake..." );

while( ( ret = mbedtls_ssl_handshake( &ssl ) ) != 0 )
{
  if( ret != MBEDTLS_ERR_SSL_WANT_READ && ret != MBEDTLS_ERR_SSL_WANT_WRITE )
  {
    mbedtls_printf( " failed\n  ! mbedtls_ssl_handshake returned -0x%x\n\n", -ret );
    goto exit;
  }
}
```

Abbildung 36 - Handschlag TLS

Bevor nun mit dem Datenaustausch begonnen werden kann, ist es sinnvoll das Serverzertifikat zu authentisieren. Dafür wird die in der folgenden Abbildung dargestellte Funktion verwendet.

```
mbedtls_printf( "  . Verifying peer X.509 certificate..." );

if( ( flags = mbedtls_ssl_get_verify_result( &ssl ) ) != 0 )
{

  mbedtls_printf( " failed\n" );
  mbedtls_x509_crt_verify_info( (char *)vrfy_buf, sizeof( vrfy_buf ), "  ! ", flags );

  mbedtls_printf( "%s\n", vrfy_buf );
}
else
{
  mbedtls_printf( " ok\n" );
}
```

Abbildung 37 - Authentisierung des Serverzertifikats

Nach diesem Schritt ist der komplette Verbindungsaufbau abgeschlossen und die Client-Server-Kommunikation kann gestartet werden. MbedTLS bietet auch dafür seine eigenen TLS-fähigen Funktionen an. In den zwei folgenden Abbildungen ist die Schreib- und die Lesefunktion abgebildet.

```
mbedtls_printf( "  > Write to server:" );

sprintf( (char *) buf, GET_REQUEST );
len = strlen((char *) buf);

while( ( ret = mbedtls_ssl_write( &ssl, buf, len ) ) <= 0 )
{
  if( ret != MBEDTLS_ERR_SSL_WANT_READ && ret != MBEDTLS_ERR_SSL_WANT_WRITE )
  {
    mbedtls_printf( " failed\n  ! mbedtls_ssl_write returned %d\n\n", ret );
    goto exit;
  }
}

len = ret;
mbedtls printf( " %d bytes written\n\n%s", len, (char *) buf );
```

Abbildung 38 - TLS Serveranfrage

```
/*
 * 7. Read the HTTP response
 */
 mbedtls_printf( "  < Read from server:" );

do
{
  len = sizeof( buf ) - 1;
  memset( buf, 0, sizeof( buf ) );
  ret = mbedtls_ssl_read( &ssl, buf, len );

  if( ret == MBEDTLS_ERR_SSL_WANT_READ || ret == MBEDTLS_ERR_SSL_WANT_WRITE )
  {
    continue;
  }

  if( ret == MBEDTLS_ERR_SSL_PEER_CLOSE_NOTIFY )
  {
    break;
  }

  if( ret < 0 )
  {
    mbedtls_printf( "failed\n  ! mbedtls_ssl_read returned %d\n\n", ret );
    break;
  }

  if( ret == 0 )
  {
    mbedtls_printf( "\n\nEOF\n\n" );
    break;
  }

  len = ret;
  mbedtls_printf( " %d bytes read\n\n%s", len, (char *) buf );
}
```

Abbildung 39 - TLS Serverantwort

Im Anschluss kann die Verbindung beendet werden. Auch dafür bietet mbedTLS eigene Funktionen an, wie in untenstehender Abbildung zu erkennen.

```
  mbedtls_ssl_close_notify( &ssl );

exit:
  mbedtls_net_free( &server_fd );

  mbedtls_x509_crt_free( &cacert );
  mbedtls_ssl_free( &ssl );
  mbedtls_ssl_config_free( &conf );
  mbedtls_ctr_drbg_free( &ctr_drbg );
  mbedtls_entropy_free( &entropy );
```

Abbildung 40 - Beenden der TLS-Verbindung

4.4.3 Testverbindung mit Webserver

Um die Funktionalität von mbedTLS zu testen, wurde eine einfache, verschlüsselte Verbindung mit einem Apache-Webserver aufgebaut. Verwendet wurden dabei die von mbedTLS mitgelieferten Schlüssel und Zertifikate. Der Programmablauf und die dazuge-

hörigen Funktionen des Verbindungsaufbaus wurden bereits ausführlich im Kapitel 4.4.2 erläutert.

Bei dieser Testverbindung wurde eine einfache HTTP-GET-Anfrage an den Webserver geschickt und auf die Reaktion bzw. Antwort gewartet. Der Inhalt der Anfrage ist in folgender Abbildung zu sehen.

```
#define GET_REQUEST "GET 192.168.6.1 HTTP/1.1\r\n\r\n"
```

Abbildung 41 - GET-Anfrage

Dabei handelt es sich um eine einfache Verbindungsanfrage, wie sie unter anderem auch Browser beim Aufruf einer Website tätigen. Ein eingebettetes System würde zwar in der Regel keine Website aufrufen, in diesem Beispiel geht es aber vorrangig darum, dass der Datenaustausch auch tatsächlich verschlüsselt erfolgt.

Über das Programm Wireshark wurde die Netzwerkkommunikation überwacht, um die Verschlüsselung der Nachrichten zu überprüfen. In der folgenden Abbildung sind der Verbindungsablauf und der Nachrichtenaustausch zu sehen.

	Source	Destination	Protocol	Length	Info
	192.168.6.2	192.168.6.1	TCP	60	49153 → 443 [SYN] Seq=0 Win=2920 Len=0 MSS=1460
1	192.168.6.2	192.168.6.1	TCP	60	[TCP Retransmission] 49153 → 443 [SYN] Seq=0 Win=2920 Len=0 MSS=1460
	192.168.6.1	192.168.6.2	TCP	58	443 → 49153 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460
	192.168.6.2	192.168.6.1	TCP	60	49153 → 443 [ACK] Seq=1 Ack=1 Win=2920 Len=0
	192.168.6.2	192.168.6.1	TLSv1.2	464	Client Hello
	192.168.6.1	192.168.6.2	TLSv1.2	773	Server Hello, Certificate, Server Key Exchange, Server Hello Done
2	192.168.6.2	192.168.6.1	TCP	60	49153 → 443 [ACK] Seq=411 Ack=720 Win=2201 Len=0
	192.168.6.2	192.168.6.1	TLSv1.2	129	Client Key Exchange
	192.168.6.1	192.168.6.2	TCP	54	443 → 49153 [ACK] Seq=720 Ack=486 Win=64165 Len=0
	192.168.6.2	192.168.6.1	TLSv1.2	105	Change Cipher Spec, Encrypted Handshake Message
	192.168.6.1	192.168.6.2	TLSv1.2	328	New Session Ticket, Change Cipher Spec, Encrypted Handshake Message
	192.168.6.2	192.168.6.1	TLSv1.2	111	Application Data
3	192.168.6.1	192.168.6.2	TLSv1.2	1354	Application Data, Application Data
	192.168.6.1	192.168.6.2	TLSv1.2	85	Encrypted Alert
	192.168.6.1	192.168.6.2	TCP	54	443 → 49153 [FIN, ACK] Seq=2325 Ack=594 Win=64057 Len=0
	192.168.6.2	192.168.6.1	TCP	60	49153 → 443 [ACK] Seq=594 Ack=2325 Win=596 Len=0
	192.168.6.2	192.168.6.1	TCP	60	49153 → 443 [ACK] Seq=594 Ack=2326 Win=595 Len=0
	192.168.6.2	192.168.6.1	TCP	60	[TCP Window Update] 49153 → 443 [ACK] Seq=594 Ack=2326 Win=2889 Len=0
	192.168.6.2	192.168.6.1	TLSv1.2	85	Encrypted Alert
	192.168.6.2	192.168.6.1	TCP	60	49153 → 443 [FIN, ACK] Seq=625 Ack=2326 Win=2889 Len=0
4	192.168.6.1	192.168.6.2	TCP	54	443 → 49153 [ACK] Seq=2326 Ack=626 Win=64026 Len=0

Abbildung 42 - TLS-Verbindung Wireshark

In der Abbildung erkennt man den Verbindungsaufbau über TCP (1), TLS-Handshake (2) inklusive Schlüssel- bzw. Zertifikattausch, Datenaustausch (3) und das Schließen der Verbindung (4). Typisch für TCP, werden die Empfangsbestätigungen nicht verschlüsselt. Da diese keine relevanten Informationen in Bezug auf den Nachrichteninhalt haben, stellt dies allerdings kein Problem dar.

In der folgenden Abbildung, ist die gesendete und verschlüsselte GET-Anfrage (vgl. Abbildung 38) zu sehen. Sie ist nicht im Klartext gesendet worden, so dass man auch nicht

den Inhalt der Nachricht erkennen kann. Somit ist die Funktionalität von mbedTLS ausreichend festgestellt.

```
0000  00 24 9b 1f 2a 27 00 50  c2 f8 0a 94 08 00 45 00   .$..*'.P ......E.
0010  00 61 00 07 00 00 ff 06  2e 3c c0 a8 06 02 c0 a8   .a...... .<......
0020  06 01 c0 01 01 bb 00 00  1b 86 05 d4 99 19 50 18   ........ ......P.
0030  07 87 52 27 00 00 17 03  03 00 34 00 00 00 00 00   ..R'.... ..4.....
0040  00 00 01 b3 02 5d 6b 0e  95 2e 4d 28 79 e1 85 19   .....]k. ..M(y...
0050  2a 8e 7f 2b 9b 6b 2a 28  3c 5a c6 41 34 e6 da b7   *..+.k*( <Z.A4...
0060  62 6b f2 15 5f e1 d8 5c  95 b4 9e f3 0f fe 5b      bk.._..\ .....[
```

Abbildung 43 - Verschlüsselte GET-Anfrage

4.5 Verbindung mit einem MQTT-Broker

Im letzten Schritt ist es vorgesehen, eine verschlüsselte Verbindung zu einem MQTT-Broker aufzubauen und ebenso den Datenaustausch zu verschlüsseln. Bei MQTT handelt es sich um ein M2M und IoT Verbindungsprotokoll. Es funktioniert nach dem publish/subscribe-Verfahren. Dabei können mehrere Clients bei einem Server, hier Broker genannt, nach erfolgreichem Verbindungsaufbau ein Thema abonnieren. Wenn nun ein anderer Client eine Nachricht veröffentlicht, erhalten alle Teilnehmer diese Nachricht, die dieses Thema abonnierten. In der folgenden Abbildung ist dieses Verfahren zu sehen. [48]

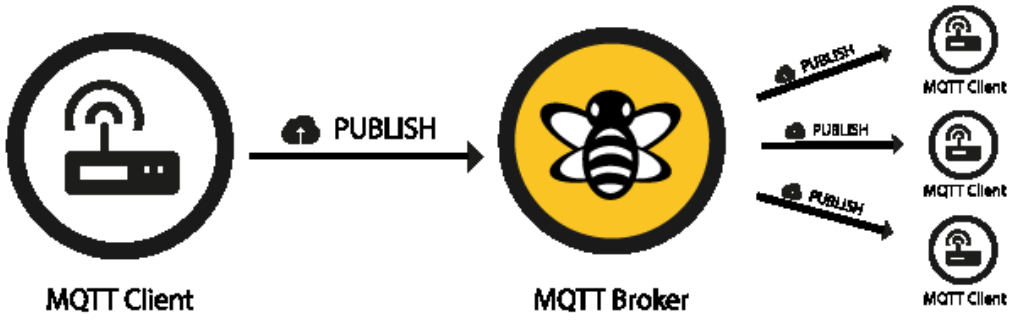

Abbildung 44 - MQTT publish/subscribe [48]

4.5.1 Unverschlüsselter Verbindungsaufbau

Um zunächst die Funktionalität und die MQTT-Verbindung zu testen, wurde vorerst ein unverschlüsselter Verbindungsaufbau vorgenommen. Die funktionsfähige Implementation eines MQTT-Clients ist bereits in einem anderen Kontext von SYS TEC ebenfalls mit einem STM32F407 im Einsatz, so dass diese übernommen werden konnte.

In den folgenden Abbildungen ist der Verbindungsaufbau, überwacht mit Wireshark, aus der Sicht des Clients abgebildet.

	Source	Destination	Protocol	Length	Info
1	192.168.6.2	192.168.6.1	TCP	60	49153 → 1884 [SYN] Seq=0 Win=2920 Len=0 MSS=1460
	192.168.6.1	192.168.6.2	TCP	58	1884 → 49153 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460
	192.168.6.2	192.168.6.1	TCP	60	49153 → 1884 [ACK] Seq=1 Ack=1 Win=2920 Len=0
2	192.168.6.2	192.168.6.1	TCP	101	49153 → 1884 [PSH, ACK] Seq=1 Ack=1 Win=2920 Len=47
	192.168.6.1	192.168.6.2	TCP	58	1884 → 49153 [PSH, ACK] Seq=1 Ack=48 Win=64240 Len=4
	192.168.6.2	192.168.6.1	TCP	78	49153 → 1884 [PSH, ACK] Seq=48 Ack=5 Win=2916 Len=24
	192.168.6.1	192.168.6.2	TCP	59	1884 → 49153 [PSH, ACK] Seq=5 Ack=72 Win=64216 Len=5
	192.168.6.2	192.168.6.1	TCP	60	49153 → 1884 [ACK] Seq=72 Ack=10 Win=2911 Len=0
3	192.168.6.1	192.168.6.2	TCP	87	1884 → 49153 [PSH, ACK] Seq=10 Ack=72 Win=64216 Len=33
	192.168.6.2	192.168.6.1	TCP	94	49153 → 1884 [PSH, ACK] Seq=72 Ack=43 Win=2878 Len=40
	192.168.6.1	192.168.6.2	TCP	54	1884 → 49153 [ACK] Seq=43 Ack=112 Win=64176 Len=0

Abbildung 45 - Verbindungsaufbau MQTT-Client

Der Verbindungsablauf erfolgt, wie in Kapitel 2.3.2.1 erläutert, über TCP (1). Nach erfolgreicher Verbindung, wird vom Client eine Subscribe-Anfrage (2) gestellt, die der Broker akzeptiert. Anschließend ist der Kommunikationskanal offen und eine Nachricht wird vom Broker an den Client geschickt (3). Der Inhalt der markierten Nachricht, ist in der folgenden Abbildung zu sehen.

```
0000  00 50 c2 f8 0a 94 00 24  9b 1f 2a 27 08 00 45 00   .P.....$ ..*'..E.
0010  00 49 12 e0 40 00 80 06  00 00 c0 a8 06 01 c0 a8   .I..@... ........
0020  06 02 07 5c c0 01 a7 05  a2 8f 00 00 19 b5 50 18   ...\.... ......P.
0030  fa d8 8d 8f 00 00 30 1f  00 11 41 6c 70 68 61 4e   ......0. ..AlphaN
0040  6f 64 65 2f 43 6f 6d 6d  61 6e 64 54 45 53 54 5f   ode/Comm andTEST_
0050  50 55 42 4c 49 53 48                              PUBLISH
```

Abbildung 46 - MQTT Publish an Client

Bei der Nachricht, gesendet von einem anderen Client an den Broker, handelt es sich um das Subscribe-Thema „AlphaNode/Comm“ und dem Inhalt „TEST_PUBLISH“.

4.5.2 Verschlüsseln der Verbindung

Im nächsten Schritt muss nun die MQTT-Kommunikation mithilfe von SSL/TLS verschlüsselt werden. Zum Abschluss dieser Arbeit, war es aber leider nicht möglich, erfolgreich einen verschlüsselten Verbindungsaufbau und Nachrichtenaustausch mit einem MQTT-Broker zu realisieren.

Die Annahme ist, dass für eine erfolgreiche TLS-Verbindung, eine Kombination aus dem Verbindungsaufbau von mbedTLS und dem publish/subscribe-Verfahren von MQTT erzeugt werden müsste. Der MQTT-Broker besitzt weiterhin standardmäßig kein eigenes Zertifikat. Daher ist es notwendig, ein Zertifikat zu erzeugen, oder eines zu erwerben.

Aufgrund der in Kapitel 4.6 beschriebenen Probleme bei dem Umstieg auf den STM32F777 verzögerte sich die Bearbeitung. Dies sorgte dafür, dass eine verschlüsselte Verbindung mit dem MQTT-Broker zum Abschluss dieser Arbeit nicht fertig gestellt werden konnte.

4.6 Umstieg auf den IoT-Chip

Alle bisher in diesem Kapitel beschriebenen Schritte wurden, mit Ausnahme von Punkt 4.5.2, erfolgreich auf dem STM32F407 umgesetzt. Mit einigen Anpassungen sollten diese auch auf dem STM32F777 durchführbar sein.

Für die Netzwerkkommunikation des STM32F7 verwendet SYS TEC einen eigenen und auf lwIP basierenden Ethernet-Treiber. Die Vorgabe ist es, dass dieser auf dem IoT-Chip verwendet werden soll. Die Konfiguration der GPIOs sowie den Registern der PHY werden in diesem Treiber bereits vorgenommen, so dass in dieser Hinsicht keine Anpassungen notwendig sind.

Bei der Implementation traten jedoch erhebliche Komplikationen mit FreeRTOS auf, welche dafür sorgten, dass eine vollständige Erfüllung der Aufgabenstellung im vorgegeben Bearbeitungszeitraum nicht möglich war. Es entstanden weiterhin erhebliche Verzögerungen durch die, im Vergleich zum STM32F4, andere Struktur des Treibers, erneute Einarbeitung in die Funktionsweise und unerwartet auftretende Probleme.

In diesem Abschnitt wird daher darauf eingegangen, welche Probleme bei dem Umstieg auf den IoT-Chip auftraten und welche Lösungsansätze verfolgt wurden.

4.6.1 Kompatibilität mit FreeRTOS

Der Treiber ist durch seinen Funktionsablauf nicht ohne weiteres für FreeRTOS geeignet, da keine der Funktionen innerhalb eines Tasks abläuft. Dadurch ist es nicht möglich, dass das RTOS arbeiten kann. Jedoch besitzt der Treiber ebenso eine Funktion, die für das Auslesen eingehender Pakete vom Netzwerkinterface verantwortlich ist, wie in Kapitel 4.3.3 erläutert. Diese Funktion wurde, durch die Ergänzung einer Endlosschleife und Abfrage eines Semaphors, zu einem Task umstrukturiert. Durch diese Anpassung ist es möglich, dass FreeRTOS in Kombination mit dem Ethernet-Treiber funktioniert.

4.6.2 Interruptprioritäten

Für seine Funktionen nutzt der Ethernet-Treiber verschiedene Interrupts, wie zum Beispiel den Rx- und Tx-Interrupt, für den Empfang und das Senden von Nachrichten. Da der Treiber bei Erstellung nicht für die Verwendung eines Echtzeitbetriebssystems gedacht war, stören die zu hoch priorisierten Interrupts die Funktionsweise des RTOS. Eine entsprechende Reduktion der Priorität musste deshalb vorgenommen werden. Ebenso sorgte der RBU-Interrupt unerwartet dafür, dass FreeRTOS nicht arbeiten konnte, weil ausschließlich nur noch Ethernet-Interrrupt-Funktionen bearbeitet wurden. Der RBU-Interrupt wurde daher deaktiviert.

4.6.3 Behandlung empfangener Nachrichtenpakete

Beim Hinzufügen des Netzwerkinterfaces musste ebenso die übergebene Input-Funktion ethernet_input ausgetauscht werden. Diese Funktion sorgt dafür, dass die Kopfdaten des empfangenen Pakets wegschnitten werden und übergibt das Ergebnis an die Funktion tcpip_input, die wiederrum die Nachricht ohne Kopfdaten verarbeitet. Das Wegschneiden der Kopfdaten übernimmt in dem neuen Treiber bereits der auslesende Task (vgl. Kapitel 4.3.3), wodurch zwei Mal in Folge Daten der Pakete entfernt und die Nachrichten unlesbar wurde. Daher wurde die übergebene Input-Funktion zu tcpip_input geändert und das doppelte Wegschneiden der Kopfdaten so verhindert. Die angepasste Erstellung des Netzwerkinterfaces ist in der folgenden Abbildung gezeigt.

```
/* add the network interface */
netif_add(&pNetIf_p, &addr, &netmask, &gw, NULL, &ethernetif_init, &tcpip_input);
```

Abbildung 47 - Anpassung Input-Funktion

Zwar funktioniert nun die Kommunikation, zum Beispiel reagiert der IoT-Chip auf Ping-Anfragen, allerdings nur innerhalb der ersten zehn Sekunden nach Neustart des Systems. Danach nahm der IoT-Chip keine Kommunikation an und reagierte auf keine Anfragen. Für dieses Verhalten konnte bis zum Abschluss der Diplomarbeit leider keine Lösung gefunden werden.

5 Auswertung und Ausblick

Ziel der vorliegenden Arbeit war es, unter Verwendung des Echtzeitbetriebssystems FreeRTOS und dem TCP/IP-Stack lwIP, auf einer SYS TEC-eigenen Referenzhardware mit einem Crypto/Hash-Prozessor, eine verschlüsselte Verbindung zu einem MQTT-Broker aufzubauen. Dabei sollte evaluiert werden, inwiefern eine verschlüsselte TCP-Kommunikation unter Verwendung der Hardwareverschlüsselung realisiert werden könnte. Ebenso sollte dabei die Laufzeitreduktion für die Übertragung von Datenblöcken, im Vergleich zu rein softwarebasierten Verschlüsselung bewertet werden.

Zu Beginn der Arbeit konnte, aufgrund von Lieferschwierigkeiten seitens ST, der STM32F7 nicht verwendet werden. Daher wurden die Umsetzung zunächst auf dem STM32F4 begonnen. Unter der Verwendung der ST-eigenen HAL-Bibliothek konnte nach der Implementation von FreeRTOS, lwIP und mbedTLS eine verschlüsselte Verbindung zu einem Web-Server aufgebaut werden.

Bei dem Wechsel von STM32F4 auf den STM32F7 sollte der verwendete Ethernet-Treiber ausgetauscht werden. Der bisher genutzte Treiber stammte aus einer Beispielanwendung von ST. Die SYS TEC-Version des Ethernet-Treibers wurde zwar bereits auf den STM32F7 angepasst und verwendet, in dieser Form war er jedoch nicht mit FreeRTOS kompatibel. Entsprechende Änderungen mussten daher vorgenommen werden. Trotz der Anpassungen konnte jedoch keine erfolgreiche Netzwerkkommunikation hergestellt werden. Der STM32F7 reagiert nur innerhalb der ersten Sekunden nach dem Start des Systems. Die Reaktionszeit steigt stetig an, bis keine Daten mehr verarbeitet werden. Trotz intensiver Tests und umfangreicher Untersuchungen, wurde für dieses Verhalten bis zum Abschluss dieser Arbeit keine Lösung gefunden. Diese Probleme verzögerten die weitere Bearbeitung erheblich. Weiterhin kann, durch die fehlende Funktionalität der Netzwerkkommunikation, keine Aussage darüber getroffen werden, wie die TCP-Kommunikation mit der Hardwareverschlüsselung erreicht wird und wieviel schneller eine hardwareunterstützte Verschlüsselung im Vergleich zur rein softwarebasierten ist.

Da ein Großteil der Zielstellung bereits auf dem STM32F4 realisiert wurde und die Treiber-Probleme nicht gelöst werden konnten, sollte die Aufgabenstellung auf dem STM32F4 beendet werden. An diesem Punkt wurde jedoch bereits zu viel Zeit in die Lösung des Treiber-Problems investiert. Eine erfolgreich verschlüsselte MQTT-Verbindung konnte daher nicht realisiert werden.

Für die zukünftige Verwendung des STM32F7 müsste folgend die Funktion des Ethernet-Treibers hergestellt werden. Bei den Untersuchungen und Tests konnte zwar keine konkrete Ursache gefunden werden, jedoch wird angenommen, dass die Fehlfunktion in Zu-

sammenhang mit der Interrupt-Verarbeitung bzw. dem Auslesen der Daten aus dem Buffer steht.

Die weiteren Implementationen sind überwiegend nur auf eine funktionsfähige Netzwerkkommunikation angewiesen. Daraus resultiert die Annahme, dass ein Großteil der Implementation vom STM32F4 übernommen werden kann. Weiterhin wäre es notwendig, unter Verwendung des Crypto/Hash-Prozessors zu ermitteln, inwiefern der Verschlüsselungsprozess tatsächlich beschleunigt werden kann.

Literatur

[1] mouser electronics
http://www.mouser.com/images/microsites/iot-wearable-fig02.jpg, verfügbar am 19.07.17, 09:53 Uhr

[2] Hackmann, Joachim: Internet of Things (IoT) in der Praxis: Industrie 4.0 ist das Internet der Ingenieure
<https://www.computerwoche.de/a/industrie-4-0-ist-das-internet-der-ingenieure,2538117>, verfügbar am 19.07.17, 09:53 Uhr

[3] Winboard: Real-Time Betriebssystem
<http://wiki.winboard.org/index.php/Real-Time_Betriebssystem>, verfügbar am 19.07.17, 10:47 Uhr

[4] Hall, Ian; Plechinger, Günter: Echtzeitbetriebssysteme: Einführung in die RTOS-Welt
<http://www.elektroniknet.de/design-elektronik/embedded/einfuehrung-in-die-rtos-welt-29528.html>, verfügbar am 19.07.17, 10:47 Uhr

[5] Rheinwerk <openbook>. „Java ist auch eine Insel“ Kapitel 12.1 Nebenläufigkeit
<http://openbook.rheinwerk-verlag.de/javainsel/javainsel_12_001.html#dodtpeed4c495-0b00-4e2b-9676-b266d43670d5>, verfügbar am 19.07.17, 11:07 Uhr

[6] FreeRTOS
<http://www.freertos.org/implementation/suspending.gif>, verfügbar 19.07.17, 11:11 Uhr

[7] elektronik-kompendium: ISO/OSI-7-Schichtmodell <https://www.elektronik-kompendium.de/sites/kom/0301201.htm>, verfügbar 19.07.17, 12:06 Uhr

[8] Wikipedia-User deadlyhappen <https://de.wikipedia.org/wiki/OSI-Modell#/media/File:ISO-OSI-7-Schichten-Modell(in_Deutsch).svg>, verfügbar am 19.07.17, 12:08 Uhr

[9] Schmeh, Klaus: Kryptografie und Public-Key-Infrastrukturen im Internet 2. Auflage, Heidelberg, dpunkt.verlag GmbH, 2001

[10] elektronik-kompendium: TCP/IP <https://www.elektronik-kompendium.de/sites/net/0606251.htm>, verfügbar 20.07.17, 14:47 Uhr

[11] ITwissen: Internet Protocol <http://www.itwissen.info/IP-Internet-protocol-IP-Protokoll.html>, verfügbar 20.07.17, 14:56 Uhr

[12] IPLocation: What is IPv6 Address? <https://www.iplocation.net/ipv6-address>, verfügbar am 21.07.17, 12:15 Uhr

[13] elektronik-kompendium: TCP-Kommunikation, <https://www.elektronik-kompendium.de/sites/net/2009211.htm>, verfügbar 20.07.17, 15:00 Uhr

[14] elektronik-kompendium <https://www.elektronik-kompendium.de/sites/net/bilder/08122713.gif>, verfügbar 20.07.17, 15:15 Uhr

[15] elektronik-kompendium <https://www.elektronik-kompendium.de/sites/net/bilder/08122715.gif>, verfügbar am 20.07.17, 15:20 Uhr

[16] elektronik-kompendium <https://www.elektronik-kompendium.de/sites/net/bilder/08122714.gif>, verfübar am 20.07.17, 15:20 Uhr

[17] mouser electronics - Wearable Devices and the Internet of Things, <http://www.mouser.de/applications/article-iot-wearable-devices/>, verfügbar am 21.07.17, 12:10 Uhr

[18] von Gagern, Stefan: Was ist das Internet der Dinge? <https://www.computerwoche.de/a/was-ist-was-im-internet-der-dinge,3213802>, verfügbar am 21.07.17, 12:13 Uhr

[19] elektronik-kompendium: Symmetrische Kryptografie (Verschlüsselung), <http://www.elektronik-kompendium.de/sites/net/1910101.htm>, verfügbar am 21.07.17, 12:45 Uhr

[20] elektronik-kompendium, <http://www.elektronik-kompendium.de/sites/net/bilder/19070411.png>, verfügbar am 21.07.17, 12:45 Uhr

[21] elektronik-kompendium: Asymmetrische Kryptografie (Verschlüsselung), <http://www.elektronik-kompendium.de/sites/net/1910111.htm>, verfügbar am 21.07.17, 12:57 Uhr

[22] elektronik-kompendium
<http://www.elektronik-kompendium.de/sites/net/bilder/19070412.png>, verfügbar am 21.07.17, 12:58 Uhr

[23] elektronik-kompendium: Hybride Verschlüsselungsverfahren
<http://www.elektronik-kompendium.de/sites/net/1910141.htm>, verfügbar am 21.07.17, 13:10 Uhr

[24] elektronik-kompendium
<http://www.elektronik-kompendium.de/sites/net/bilder/09080714.gif>, verfügbar am 21.07.17, 13:10 Uhr

[25] STMicroelectronics: STM32F7 Series,
<http://www.st.com/en/microcontrollers/stm32f7-series.html?querycriteria=productId=SS1858>, verfügbar am 21.07.17, 13:30 Uhr

[26] STMicroelectronics: STM32 32-bit ARM Cortex MCUs,
<http://www.st.com/en/microcontrollers/stm32-32-bit-arm-cortex-mcus.html?querycriteria=productId=SC1169>, verfügbar am 21.07.17, 13:35 Uhr

[27] FreeRTOS,
<http://www.freertos.org/>, verfügbar am 21.07.17, 14:55 Uhr

[28] LwIP,
<https://savannah.nongnu.org/projects/lwip/>, verfügbar am 21.07.17, 14:29 Uhr

[29] mbedTLS: SSL Library from mbedTLS: Easy to use open source SSL in C, <https://tls.mbed.org/ssl-library>, verfügbar am 21.07.17, 14:27 Uhr

[30] ARM: ARM buys Leading IoT Security Company Offspark as it Expands its mbed Platform <https://www.arm.com/about/newsroom/arm-buys-leading-iot-security-company-offspark-as-it-expands-its-mbed-platform.php>, verfügbar am 21.07.17, 14:27 Uhr

[31] elektronik-kompendium: Verschlüsselung / Chiffrierung <http://www.elektronik-kompendium.de/sites/net/1907041.htm>, verfügbar am 21.07.17, 13:27 Uhr

[32] Winboard: Scheduler <http://wiki.winboard.org/index.php/Scheduler>, verfügbar am 22.07.17, 14:11 Uhr

[33] SYS TEC electronic GmbH: System Manual PLCcore-F407 User Manual Version 5, Oktober 2015

[34] ST: STM32Cube initialization code generator <http://www.st.com/en/development-tools/stm32cubemx.html>, verfügbar am 01.08.17, 14:12 Uhr

[35] FreeRTOS: Running the RTOS on an ARM Cortex-M Core <http://www.freertos.org/RTOS-Cortex-M3-M4.html>, verfügbar am 06.08.17, 14:23 Uhr

[36] FreeRTOS: Static vs Dynamic Memory Allocation <http://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html>, verfügbar am 06.08.17, 14:23 Uhr

[37] FreeRTOS: Memory Management
<http://www.freertos.org/a00111.html>, verfügbar am 06.08.17, 14:23 Uhr

[38] LwIP: Raw/native API
<http://lwip.wikia.com/wiki/Raw/native_API>, verfügbar am 08.08.17, 12:34 Uhr

[39] LwIP: Netconn API
<http://lwip.wikia.com/wiki/Netconn_API>, verfügbar am 08.08.17, 12:36 Uhr

[40] LwIP: Application API layers
<http://lwip.wikia.com/wiki/Application_API_layers>, verfügbar am 09.08.17, 12:34 Uhr

[41] FreeRTOS: xTaskCreate
<http://www.freertos.org/a00125.html>, verfügbar am 08.08.17, 12:34 Uhr

[42] Portolani, M., Arregoces, M.: Data Center Fundamentals, First Print, Indianapolis, Cisco Press, December 2003

[43] elektronik-kompendium: Verbindungsaufbau SSL/TLS
<http://www.elektronik-kompendium.de/sites/net/bilder/09022816.gif>, verfügbar am 08.08.17, 12:30 Uhr

[44] elektronik-kompendium: SSL - Secure Socket Layer
<www.elektronik-kompendium.de/sites/net/0902281.htm>, verfügbar am 08.08.17, 13:33 Uhr

[45] TreckWiki: Introduction to BSD Sockets <http://wiki.treck.com/Introduction_to_BSD_Sockets>, verfügbar am 12.08.17, 13:28 Uhr

[46] SYS TEC electronics GmbH – IoT Chip SE <http://www.systec-electro-nic.com/uploads/12/a7/12a75a59d6877a817809b23d03a526c9/IoT_Chip-1_450x450.png>, verfügbar am 12.08.17, 12:23 Uhr

[47] HiveMQ: MQTT Essentials Part 4: MQTT Publish, Subscribe & Unsubscribe <http://www.hivemq.com/blog/mqtt-essentials-part-4-mqtt-publish-subscribe-unsubscribe>, verfügbar am 20.08.17, 14:34 Uhr

[48] HiveMQ <http://www.hivemq.com/wp-content/uploads/publish_flow.png>, verfügbar am 20.08.17, 14:35 Uhr

[49] Heise: Entropie <https://www.heise.de/glossar/entry/Entropie-397939.html>, verfügbar am 22.08.17, 12:41 Uhr

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Mittweida, den 31.08.2017

Christian Schuster